2013

Design of an Open-Source SATA Core for Virtex-4 FPGAs

Cory Gorman
University of Massachusetts Amherst, corywgorman@gmail.com

DESIGN OF AN OPEN-SOURCE SATA CORE FOR VIRTEX-4 FPGAS

A Thesis Presented

by

CORY W. GORMAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2013

ELECTRICAL AND COMPUTER ENGINEERING

DESIGN OF AN OPEN-SOURCE SATA CORE FOR VIRTEX-4 FPGAS

A Thesis Presented

By

CORY W. GORMAN

Approved as to style and content by:

____
Russell G. Tessier, Chair

____
Paul Siqueira, Member

____
David Irwin, Member

____
C. V. Hollot, Department Head
Electrical and Computer Engineering

ABSTRACT

DESIGN OF AN OPEN-SOURCE SATA CORE FOR VIRTEX-4 FPGAS

SEPTEMBER 2013

CORY W. GORMAN, B.S., UNIVERSITY OF MASSACHUSETTS AMHERST

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell G. Tessier

Many hard drives manufactured today use the Serial ATA (SATA) protocol to communicate with the host machine, typically a PC. SATA is a much faster and much more robust protocol than its predecessor, ATA (also referred to as Parallel ATA or IDE). Many hardware designs, including those using Field-Programmable Gate Arrays (FPGAs), have a need for a long-term storage solution, and a hard drive would be ideal. One such design is the high-speed Data Acquisition System (DAS) created for the NASA Surface Water and Ocean Topography mission. This system utilizes a Xilinx Virtex-4 FPGA. Although the DAS includes a SATA connector for interfacing with a disk, a SATA core is needed to implement the protocol for disk operations.

In this work, an open-source SATA core for Virtex-4 FPGAs has been created. SATA cores for Virtex-5 and Virtex-6 devices were already available, but they are not compatible with the different serial transceivers in the Virtex-4. The core can interface with disks at SATA I or SATA II speeds, and has been shown working at rates up to 180MB/s. It has been successfully integrated into the hardware design of the DAS board so that radar samples can be stored on the disk.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

SATA (Serial ATA) is one of the current technologies for high speed data transfer between a computer and a peripheral. Nearly all consumer hard drives manufactured today use SATA [1]. It replaced the older Parallel ATA (PATA, also known as IDE), which was the dominant standard for many years. SATA has some significant advantages over the older technology, including higher speeds and greater reliability. The fastest transfer rate possible with PATA is 133MB/s, while the first generation of SATA is capable of 150MB/s [2].

A system that has a large amount of data that needs to be stored quickly will likely find a SATA hard drive to be a viable solution. The NASA SWOT radar Data Acquisition System (DAS) is one such system.

The Surface Water and Ocean Topography (SWOT) mission seeks to monitor the levels of various water bodies using a satellite-mounted radar system. To achieve this goal, an FPGA-based board using a Ka-band radar interferometer (KaRIN) was designed at the University of Massachusetts Amherst [3]. This board features a Xilinx Virtex-4 FPGA and connectors for various high speed peripherals, including SATA-compatible drives. In the final version of the board, all components will be radiation-hardened for use in space [4].

This DAS board is capable of acquiring samples at a rate of 3 Giga-samples per second. For some applications, it may be desirable to store the acquired data for later analysis. However, the amount of data would quickly exceed the on-board storage capacity of the FPGA (only 9,936 Kb—see Chapter 5 for more details), so it is necessary to store the data on a dedicated storage device such as a hard drive. To do so, one of the industry-standard hard drive interfaces must be used. Thus, SATA seems to be a suitable choice, since it features high throughput for storage, as shown in Table 1.

SATA Version	**Maximum Speed**
SATA Generation 1 (SATA I)	1.5Gb/s, 150MB/s
SATA Generation 2 (SATA II)	3.0Gb/s, 300MB/s
SATA Generation 3 (SATA III)	6.0Gb/s, 600MB/s

Table 1: SATA Generations and Speeds

The SATA protocol uses a layered approach, wherein each layer uses services of the layer below it and presents services to the layer above it. At the highest level, a fairly simple Read/Write interface is presented to applications wishing to store data, while the lower layers do many complex transformations, synchronization, and hand-shaking. The architecture of SATA will be discussed in more detail shortly.

In this thesis, we have developed a Virtex4-compatible SATA host controller. This module can store data from the radar at high speeds, while implementing the full SATA protocol stack to maintain compatibility with available drives. The core will also be released as open-source, so that other designs with Virtex-4 devices can benefit from it. It will be the first open-source Virtex-4 SATA core available.

CHAPTER 2

BACKGROUND

2.1 The DAS Board for NASA

The SATA controller developed for this thesis will be used in a radar data acquisition board, previously designed here at UMass Amherst. This prototype board is meant to be used as part of a Data Acquisition System, or DAS. A DAS is designed to measure, process, and analyze a quantity of interest. The board was designed with real-time acquisition and processing in mind.

The DAS board was designed for NASA's Surface Water and Ocean Topology (SWOT) mission. The goal of this mission is to monitor water levels and circulation characteristics of Earth's oceans [4]. The system uses a Ka-Band Radar Interferometer (KaRIN) to make measurements, which are fed into high-speed Analog-to-Digital Converters (ADCs) on the board. The digital data can then be processed, stored, or fed to another system over a compact PCI (cPCI) interface.

The board contains two Xilinx Virtex-4 Field Programmable Gate Arrays (FPGAs). An FPGA is a reprogrammable chip that can implement any sort of digital logic. FPGAs are flexible, and they allow for high-speed data processing through specialization of hardware and parallelization. One of the Virtex-4s is used to handle communication over cPCI, and the other is used for acquiring and processing data (the “Data FPGA”, see Figure 2).

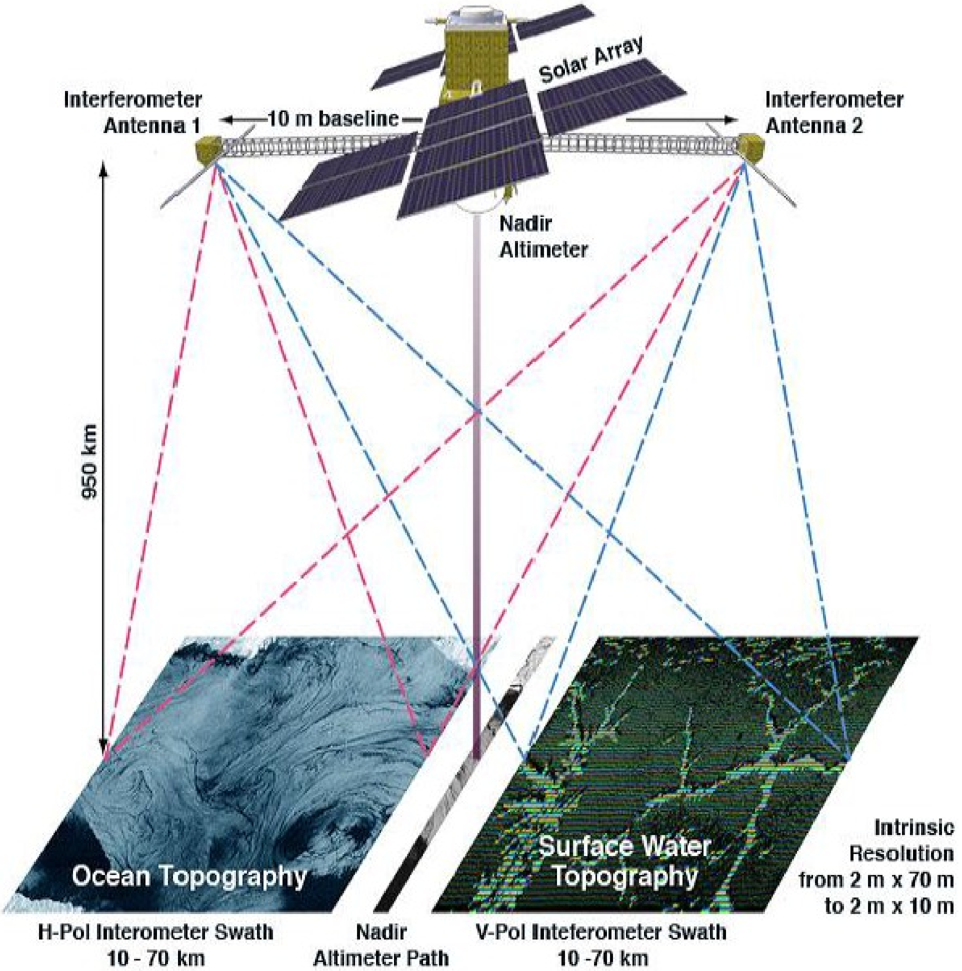


Figure 1: SWOT System

Figure 1 is from [4].

The Data FPGA on the board is connected to a SATA interface connector. Using its built-in RocketIO Multi-Gigabit Transceiver (MGT), this FPGA can communicate using a variety of high-speed protocols. One of the supported protocols is SATA, which allows for a large amount of data to be stored for later use on a hard drive. Due to the parallelizable nature of the FPGA, this data storage capability can be added without slowing down the other processing.

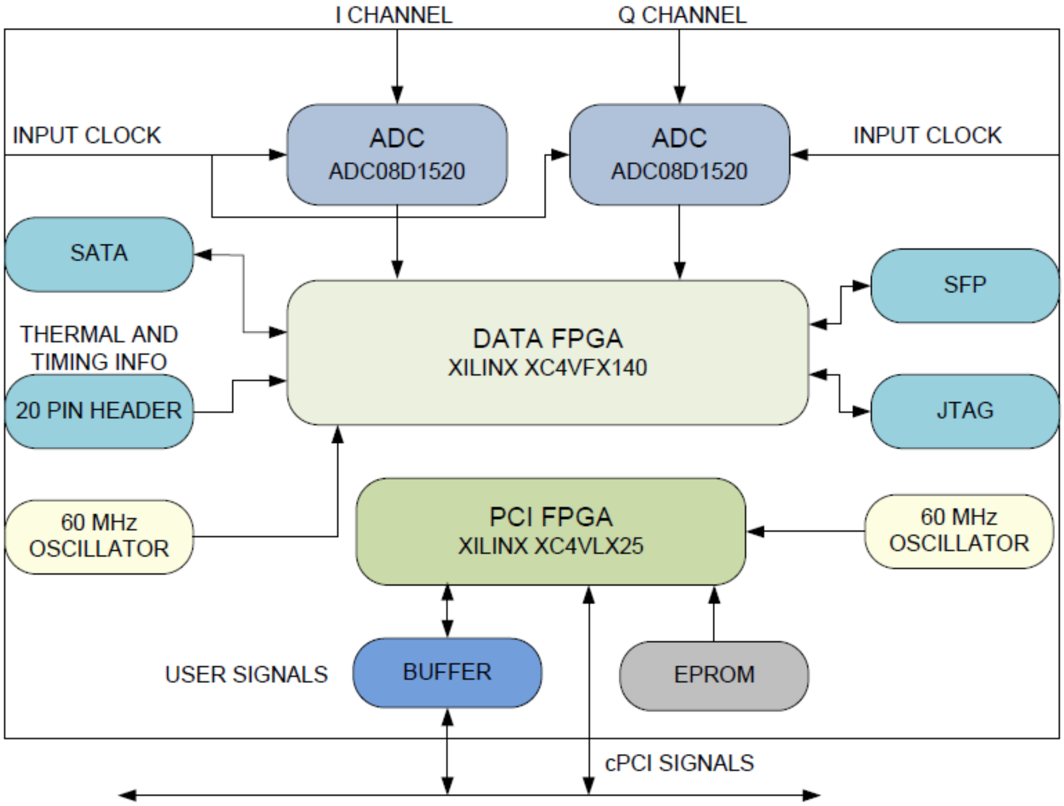


Figure 2: DAS Block Diagram

Figure 2 is from [4].

In addition to the DAS Board, the ML405 Evaluation Board is used in the work [5]. This is a development board created by Xilinx that contains a Virtex-4 FPGA, and is the primary development board for this project. Because this board is mass-produced, it is significantly less expensive than the DAS board; therefore it would be less of a setback if it were damaged in any way. Also, it has two SATA connectors instead of one, making loopback debugging possible. This has been useful during development.

2.2 SATA Overview

Serial ATA is a peripheral interface created in 2003 to replace Parallel ATA, also known as IDE. Hard drive speeds were getting faster, and would soon outpace the capabilities of the older standard—the fastest PATA speed achieved was 133MB/s, while SATA began at 150MB/s and was designed with future performance in mind [2]. Also, newer silicon technologies used lower voltages than PATA's 5V minimum. The ribbon cables used for PATA were also a problem; they were wide and blocked air flow, had a short maximum length restriction, and required many pins and signal lines [2].

SATA has a number of features that make it superior to Parallel ATA. The signaling voltages are low and the cables and connectors are very small. SATA has outpaced hard drive performance, so the interface is not a bottleneck in a system. It also has a number of new features, including hot-plug support.

SATA is a point-to-point architecture, where each SATA link contains only two devices: a SATA host (typically a computer) and the storage device. If a system requires multiple storage devices, each SATA link is maintained separately. This simplifies the protocol and allows each storage device to utilize the full capabilities of the bus simultaneously, unlike in the PATA architecture where the bus is shared.

To ease the transition to the new standard, SATA maintains backward compatibility with PATA. To do this, the Host Bus Adapter (HBA) maintains a set of *shadow registers* that mimic the registers used by PATA. The disk also maintains a set of these registers. When a register value is changed, the register set is sent across the serial line to keep both sets of registers synchronized. This allows for the software drivers to be agnostic about the interface being used.

SATA uses a layered architecture, depicted in Figure 3.The highest layer is the Application Layer, which represents the software using the SATA device. Below that is the Command Layer, which triggers series of Transport Layer actions to implement a PATA command. Next is the Transport Layer, which handles creating and formatting Frame Information Structures (FISes), and the valid sequences of FISes. Beneath that is the Link Layer, which encodes the FISes, handles control signals, and checks for FIS integrity. The lowest layer is the Physical Layer, which handles the transmission and reception of the actual electrical signal and maintains alignment. It also takes care of establishing the link, using what is known as Out-of-band (OOB) signaling.

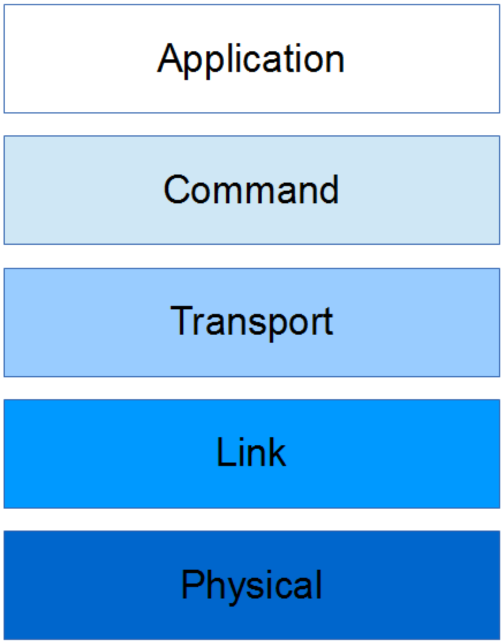


Figure 3: SATA Layer Architecture

Each layer provides services to the layer above it. This allows for each layer to "abstract away" the details of the layers below it and simplify the design process. The layers will be discussed in more depth shortly.

2.2.1 Notes on Terminology

When discussing SATA, there are multiple words that can refer to the same thing, and words could have different meanings in other contexts. To avoid ambiguity, in this document, we will try to be consistent in the use and meaning of the following terms.

Dword: Although this term is typically used in the context of a particular processor or processor family, here it refers to 32 bits of data, or 4 bytes. This is consistent with other SATA literature, such as [2] and [6]. However, note that a Dword is encoded as 40 bits while on the line. Despite the size change, this is still referred to as a "Dword" because the encoded data is never manipulated directly, and once decoded, will again be 32 bits.

Core, Host Bus Adapter (HBA): This refers to the SATA design being presented in this work. That is, the hardware that interfaces with a disk and handles the SATA protocol.

Host: This refers to the system that is interfacing with the disk, and includes the HBA. An example of a host would be a PC, or the DAS board. Since the SATA protocol is asymmetric, "Host" can also refer to the host's side of the protocol.

Disk, device: This refers to the hard drive with which we are communicating. Although disk is unambiguous, device could refer to any number of things, including a Virtex-4 device. In this work, "device" refers to the hard disk, unless context indicates otherwise.

Frame Information Structure (FIS): A Frame Information Structure, or FIS, is a single data payload that is sent over the SATA link. These are analogous to "packets" in network terminology. There are multiple types of FISes, and all of them are wrapped by Start of Frame (SOF) and End of Frame (EOF) primitives. The protocol defines valid sequences of FISes for data transfer. One or more of these FISes will be Data FISes, that actually contain the data to be read or written. The maximum size of a single FIS is 8KB.

CHAPTER 3

SATA DETAILS

3.1 Physical Layer

The physical layer is the lowest layer of the SATA protocol stack. It handles the electrical signal being sent across the cable. The physical layer also handles some other important aspects, such as resets and speed negotiation.

SATA uses low-voltage differential signaling (LVDS). Instead of sending 1's and 0's relative to a common ground, the data being sent is based on the difference in voltage between two conductors sending data. In other words, there is a TX+ and a TX- signal. A logic 1 corresponds to a high TX+ and a low TX-; and vice versa for a logic 0. SATA uses a ±125mV voltage swing [2][6].

This scheme was chosen for multiple reasons. For one, it improves resistance to noise. A source of interference will likely affect both conductors in the same way, since they are parallel to each other. However, a change in voltage on both conductors does not change the difference between them, so the signal will still be easily recovered. Low-voltage differential signaling also reduces electromagnetic interference (EMI), and the lower signaling voltages means that less power is used [2].

3.1.1 Out-of-Band Signaling

As stated earlier, the physical layer is also responsible for link initialization and resets. But how can a host and a device communicate to initialize the link if they don't have a link with which to communicate? The scheme that SATA uses is called out-of-band (or OOB) signaling.

Under this scheme, it is assumed that the host and the device can detect the presence or absence of a signal, even if they cannot yet decode that signal. OOB signals are essentially that—whether or not an in-band signal is there. By driving TX+ and TX- to the same common voltage (so not a logic 1 or a logic 0), one party can transmit an OOB "lack of signal."

Link initialization is performed by sending a sequence of OOB primitives, which are defined patterns of signal/no-signal. There are three defined primitives: COMRESET, COMINIT, and COMWAKE. Each primitive consists of six "bursts" of a present signal, with idle time in between. The times of each burst are defined in terms of "Generation 1 Unit Intervals" (U), which is the time to send 1 bit at the SATA I rate of 1.5Gb/s, or 666 ps [2].

Table 2 shows the definitions of the primitives. There are also fairly loose tolerances defined for each signal [6]. Note also that COMRESET and COMINIT have the same definition—the only difference is that COMRESET is sent by the host, and COMINIT is sent by the device.

OOB Signal	Burst Length	Inter-burst Idle Time
COMRESET	106ns (160U)	320ns (480U)
COMINIT	106ns	320ns
COMWAKE	106ns	106ns

Table 2: OOB Primitive Definitions

The COMRESET signal, sent by the host, is used to reset the link. Following a COMRESET, the OOB initialization sequence is performed again. COMRESET can also be sent repeatedly to hold the link in a reset state.

3.1.1.1 The OOB Sequence

The initialization state machine for the host follows this sequence to establish communications with the disk. This sequence is illustrated in Figure 4.

First, a COMRESET is sent. The host then waits for a COMINIT from the device. If no COMINIT is received, the host can send more COMRESETs until it receives one, and assume that no device is connected until it does. After receiving COMINIT, the host is given time to optionally calibrate its receiver and transmitter. For example, it may be necessary to adjust signal parameters or termination impedances. The host then sends a COMWAKE to the device, and expects the same in return. After this, the host waits to receive an ALIGN primitive (an in-band signal which will be explained shortly). Meanwhile, it sends a “dial-tone” to the device: an alternating pattern of 1's and 0's. This was intended as a cost-saving feature, so that disks with cheap oscillators could instead use the dial-tone as a reference clock for locking [2].

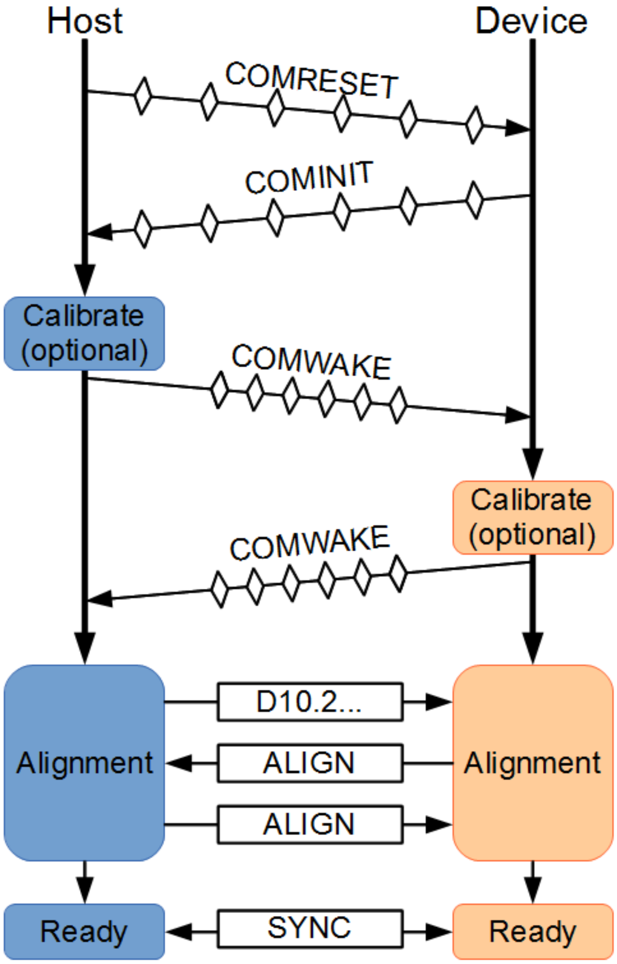


Figure 4: OOB Initialization Sequence

It is also at this stage that speed negotiation is performed. The device will send ALIGN primitives at the fastest speed it supports, and wait for the host to acknowledge them. If it does not receive an acknowledgment, then it tries the next lowest speed, and so on until an agreement is found. Alternatively, if the host supports faster speeds than the device, then ALIGN primitives it receives will appear “stretched”; the host can then slow down to accommodate.

When the host receives valid ALIGN primitives, it sends ALIGNs back to acknowledge. Both parties then send SYNC or other non-ALIGN primitives, and the link is ready.

3.1.2 8b/10b Encoding

The Physical Layer also handles encoding the data before sending it. The scheme used in SATA is 8b/10b encoding, which is also used in PCI Express, USB 3.0, and many other high speed protocols [6]. 8b/10b Encoding has a number of properties that make it useful for this purpose.

One primary function of 8b/10b encoding is clock recovery. Under this scheme, there are never more than five ones or zeros in a row. In other words, there are many bit transitions in the data stream. This allows the receiver to recover the clock using a PLL or by oversampling the data. This is important for serial data, as otherwise a stream of 12 ones in a row, for example, could be interpreted as 11 or 13 ones instead.

The encoding of data maps each byte to a 10-bit character, instead of an 8-bit one. Only 10-bit characters that have enough transitions are used. Also, the scheme tries to maintain DC Balance, and uses the 10-bit patterns with an equal number of ones and zeros. However, there are not enough of these to accommodate the 256 possible values of a byte, so also those patterns with 6 zeros and 4 ones (or vice versa) are used [2].

The encoder keeps track of the running disparity to maintain DC Balance. The running disparity changes each time an uneven pattern is sent. For example, if a 10-bit character with 6 zeros and 4 ones was just sent, the running disparity is now negative. The next character therefore must have positive disparity (4 zeros and 6 ones) or neutral disparity (5 and 5). Thus, many of the bytes actually have two encodings—one positive and the other negative. The current running disparity determines which encoded value to use. Running disparity also acts as a means to detect transmission errors.

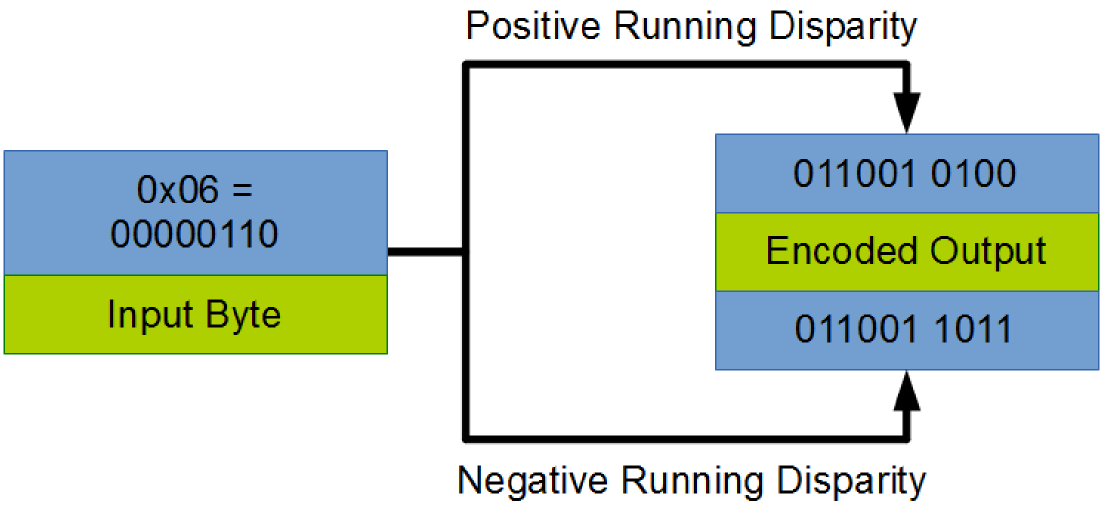


Figure 5: 8b/10b Encoding Example
In this encoding example, note that the number of ones and zeros in the output is different depending on the current disparity. Both of the encodings correspond to the same data byte (0x06, or D6.0 in the encoding table).

3.1.2.1 The Comma and the ALIGN primitive.

In addition to the 256 valid data encodings for each byte (referred to as Dx.x symbols), there are also special control symbols that can be sent. These do not correspond to a data byte and are referred to as Kx.x symbols, or K characters. SATA uses two K characters: K28.3 and K28.5. The first is used to distinguish link layer primitives, and the other is the comma character.

The comma is a special character that is used to determine byte alignment in the data stream. We've already discussed how 8b/10b encoding provides enough transitions in the stream to recover a clock (essentially providing *bit* alignment), but it would not be possible to decode the actual characters being sent if it is not known where they begin and end.

The comma is a special character because it is the only place in the data stream where there are five zeros or five ones in a row (depending on disparity), followed by two bits of the opposite. Thus, the receiver can detect this unique pattern and know that it is a comma, and therefore find the 10-bit boundaries between the symbols.

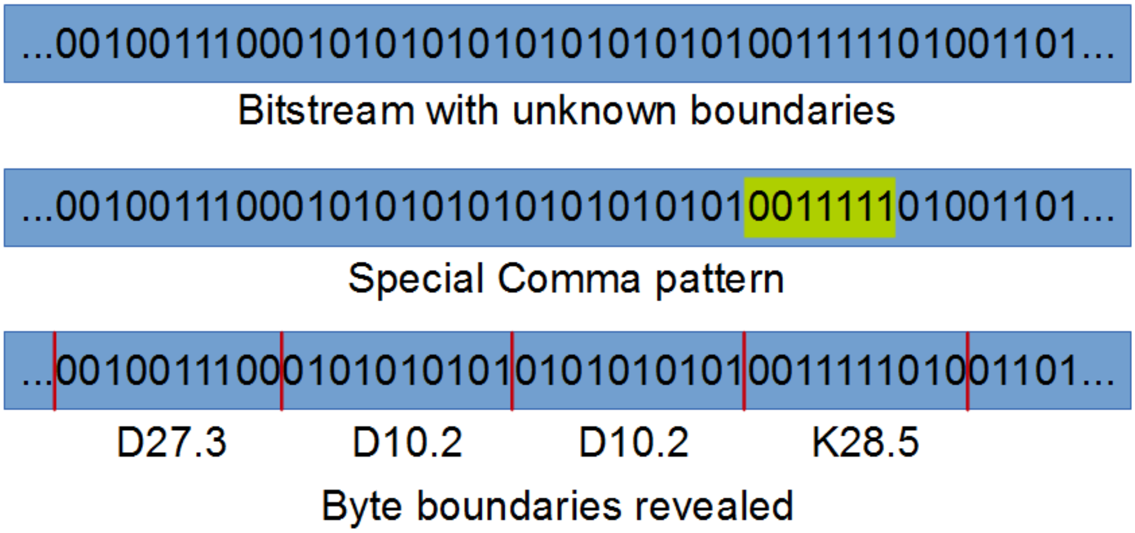


Figure 6: Comma Alignment Example

In SATA, the comma is used as part of the ALIGN primitive. Link Layer primitives, which will be discussed shortly, are 4 bytes long and always begin with a K character. The ALIGN is the only one to contain the comma, K28.5. That is why it is used as part of the link initialization procedure, so that byte boundaries can be determined before attempting to send data.

The SATA protocol also specifies that at least two ALIGNs must be sent every 256 Dwords, and they must be sent in pairs. This happens even when data is being sent. This ensures that the byte boundary is not lost, and both the host and the disk must send these ALIGNs. It also acts as a way to manage small frequency differences between the sender and receiver. For example, if the sender's clock is running a bit faster than the receiver's, the receiver's buffer may eventually overflow. Since ALIGNs are sent periodically and they are not data-important, they can be dropped to prevent this from occurring [2].

3.1.3 Spread-Spectrum Clocking

To further reduce EMI, the SATA specification requires that a receiver be able to lock to a bitstream that uses spread-spectrum clocking (SSC). SSC is a scheme where-in the line rate does not stay constant, but varies slightly over time. This spreads the emissions over a wider frequency range [2]. The transceivers on the Virtex-4 are able to receive SSC signals, but does not use it when transmitting [12].

3.2 Link Layer

The link layer is the next layer and is directly above the physical layer. This layer is responsible for encapsulating data payloads and manages the protocol for sending and receiving them. A data payload that is sent is called a *Frame Information Structure* (FIS). The link layer also provides some other services for ensuring data integrity, handling flow control, and reducing EMI.

The host and the disk each have their own transmit pair in a SATA cable, and theoretically data could be sent in both directions simultaneously. However, this does not occur. Instead, the receiver sends “backchannel” information to the sender that indicates the status of the transfer in progress. For instance, if an error were to be detected mid-transmission, such as a disparity error, the receiver could notify the sender of this.

The link layer uses a set of defined *Link Layer Primitives* to perform these functions. Primitives are each 4 Dwords long and start with the control character K28.3 (except for ALIGN, as discussed above). The following table lists most of the defined primitives and their value in hexadecimal before encoding. The usage of these will be discussed in more detail.

Primitive	Hex Representation
ALIGN	0x7B4A4ABC
SYNC	0xB5B5957C
X_RDY	0x5757B57C
R_RDY	0x4A4A957C

Primitive	Hex Representation
SOF	`0x3737B57C`
R_IP	`0x5555B57C`
HOLD	`0xD5D5AA7C`
HOLD_ACK	`0x9595AA7C`
EOF	`0xD5D5B57C`
WTRM	`0x5858B57C`
R_OK	`0x3535B57C`
R_ERR	`0x5656B57C`
CONT	`0x9999AA7C`

Table 3: Link Layer Primitives

ALIGN: This primitive, as discussed in the previous section, allows the receiver to determine the byte boundaries in the data stream. A pair of them is sent at least every 256 Dwords regardless of what state the link layer is in.

SYNC: SYNC is used to indicate that the line is idle. When frames are not being sent, both the host and the disk will send this primitive. This primitive also has a special function called the "SYNC Escape." If the host sends a SYNC, the line is forced to go idle, terminating all current transfers [2]. The disk must respond SYNC. This way, if the host needs to issue a soft reset, it can do so.

X_RDY: This primitive indicates that there is data that is ready to be sent. It will be sent repeatedly by the disk or host until it is acknowledged. If both parties are simultaneously sending X_RDY, it is expected that the host will back down [2].

R_RDY: Indicates that the party is ready to receive a FIS. This primitive is used to acknowledge X_RDY, or can be sent preemptively if a transfer is expected.

SOF: A primitive that signals the start of a FIS (Start of Frame). The next Dwords sent after this are data.

R_IP: Receive In Progress. This is a backchannel primitive that is used by a receiver to indicate that it is currently receiving the FIS.

HOLD: The HOLD primitive is used for flow control management, which will be discussed in more detail shortly.

HOLD_ACK: Acknowledges a HOLD.

EOF: A primitive that signals the end of a FIS. No more data will be sent until a new FIS transfer is started. It also indicates that the previous Dword was the CRC.

WTRM: Waiting for Termination. This is sent repeatedly after EOF by the sender of a FIS. It indicates that the sender is waiting for acknowledgment of the frame.

R_OK: This primitive is sent by the receiver to indicate that the FIS was received correctly, and that the CRC was correct.

R_ERR: This primitive indicates that there was an error with the reception of the FIS. Most likely, the CRC was incorrect. However, it could also indicate a parity error.

CONT: The CONT primitive is used to reduce EMI created by primitives. There are many times where the same primitive is sent repeatedly, and this would cause certain frequencies to have more EMI noise [2]. The CONT primitive eliminates that problem by using pseudo-randomly-generated garbage data. If a sender would send many repeated primitives, instead it can send CONT. The receiver should then treat the CONT, and all the following random data, as if the original primitive was still being sent. This continues until a new valid primitive is received (none of the junk data are K characters). For example, a sender may send SYNC, SYNC, CONT, XXXX, XXXX, …., X_RDY. The CONT indicates that the receiver should "pretend" that SYNCs are still being sent, up until the next valid primitive (X_RDY). By using garbage data instead of repeated primitives, the EMI is distributed across a broader spectrum.

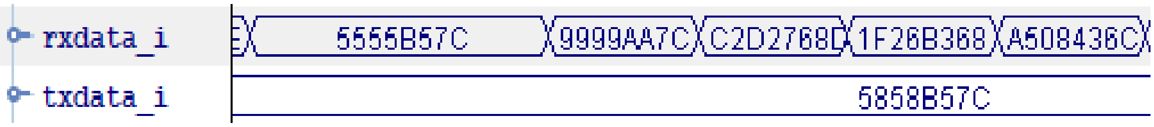


Figure 7: Link Layer Primitives in Action

In this screen capture from the Chipscope debugging tool, we see that the host is sending WTRM while the disk sends R_IP. It then sends CONT followed by some garbage data, which should be treated as a continued R_IP.

PMREQ_P/PMREQ_S/PMACK/PMNAK: These primitives facilitate power management. However, they are not implemented in this work nor are they necessary for correct SATA operation. Thus, they will not be discussed further. For more information regarding these primitives, see [2] or [6].

A typical FIS transfer happens as follows. The sender indicates that they have data to send using X_RDY. The sender then waits for R_RDY from the receiver. The sender then sends a (single) SOF, followed by the data to be sent. When the receiver sees the SOF, it will switch from sending R_RDY to R_IP. Once all of the data in the FIS has been sent, the CRC is sent, followed by EOF. The sender then starts sending WTRM until it gets R_OK, R_ERR, or SYNC. The latter two indicate an error, with SYNC meaning a protocol or unknown error. The receiver, upon getting EOF, checks the CRC, which it knows is the previous Dword. It then replies either R_OK or R_ERR. The sender acknowledges the R_OK or R_ERR by sending SYNC. The receiver then also sends SYNC, the line has returned to idle, and the transfer is complete. This process is illustrated in Figure 8.

3.2.1 Flow Control

As stated before, HOLD and HOLD_ACK are the primitives used for flow control. They are used in two situations to temporary pause the transmission of data in the middle of a FIS.

The first situation is if the receiver's buffer is getting full and can't accept any more data. For example, this could happen if a hard drive cannot write data as fast as the protocol allows. The receiver would then change from sending R_IP to HOLD. The sender would then pause the sending of data and respond with HOLD_ACK. When there is once again enough room in the buffer, the receiver sends R_IP again, and the sender can resume sending the data.

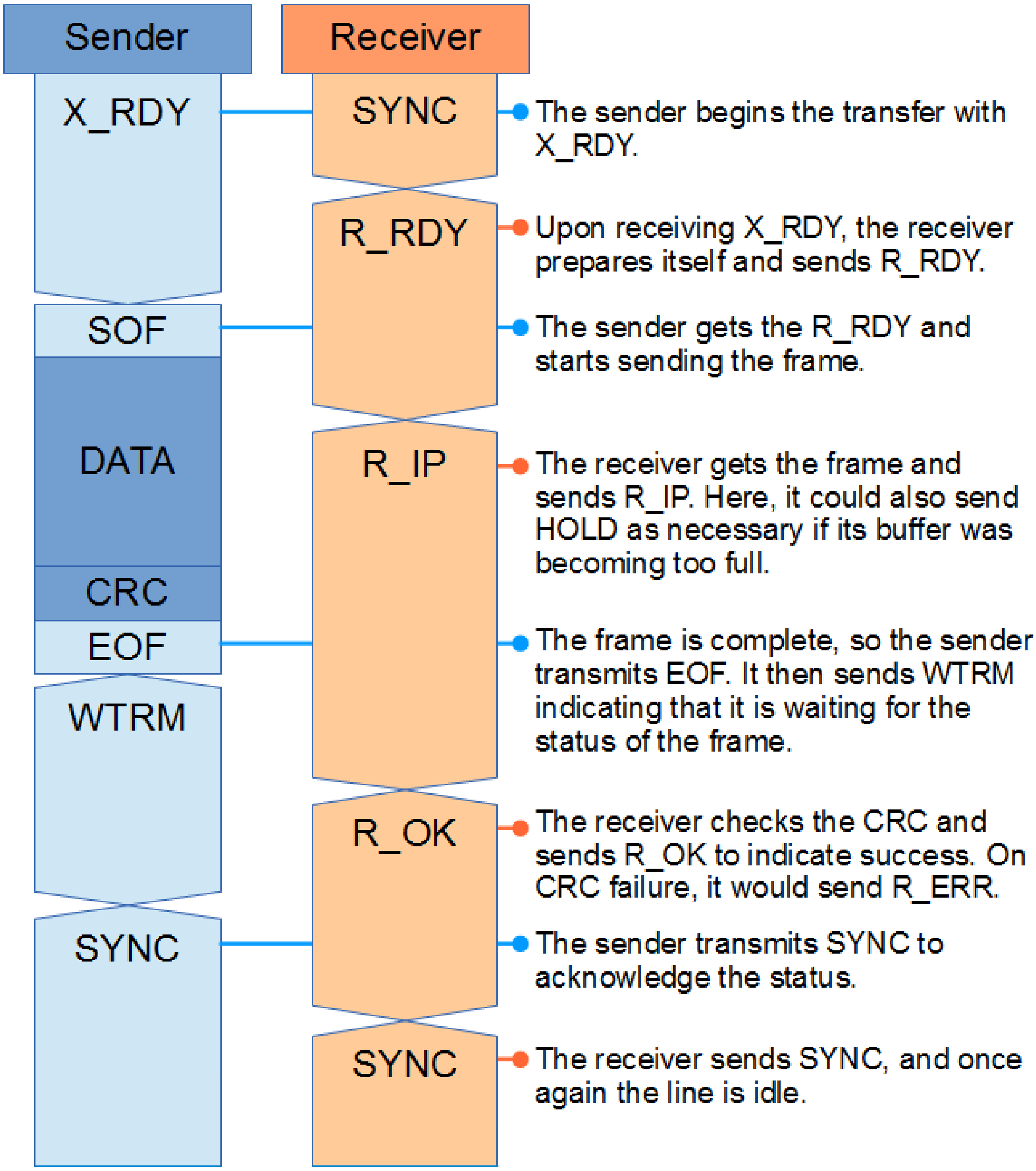

Sender
Receiver
X_RDY
SYNC
The sender begins the transfer with X_RDY.
R_RDY
Upon receiving X_RDY, the receiver prepares itself and sends R_RDY.
SOF
The sender gets the R_RDY and starts sending the frame.
DATA
R_IP
The receiver gets the frame and sends R_IP. Here, it could also send HOLD as necessary if its buffer was becoming too full.
CRC
EOF
The frame is complete, so the sender transmits EOF. It then sends WTRM indicating that it is waiting for the status of the frame.
WTRM
R_OK
The receiver checks the CRC and sends R_OK to indicate success. On CRC failure, it would send R_ERR.
SYNC
The sender transmits SYNC to acknowledge the status.
SYNC
The receiver sends SYNC, and once again the line is idle.

Figure 8: FIS Transfer

The second situation occurs when the transmitter is waiting for more data to send. In this case, the sender sends HOLD until it is ready to continue to send data. Once again, HOLD_ACK is sent in reply.

Of course, these primitives do not travel down the cable instantaneously. There is a delay between the time that a HOLD is sent and the time that the HOLD is received. But this could lead to data loss if the sending party was not yet aware of the requested HOLD and continued to send data. Thus, the protocol specifies a maximum delay, referred to as the maximum signal latency or the HOLD latency. This latency includes not only the time on the wire, but also the time to decode, interpret, and react to the HOLD.

The HOLD latency is specified as the time to send 20 Dwords [2]. Thus, a receiver can send a HOLD when there are 20 Dwords of space left in its buffer and no data will be lost. Before 20 more Dwords of data arrive, the sender will have switched to HOLD_ACK.

3.2.2 CRC

SATA uses a Cyclic Redundancy Check (CRC) on each and every FIS to ensure data integrity. The CRC used is CRC-32, the same that is used for Ethernet and some other protocols [6]. This CRC can reliably detect up to two bit errors on data blocks as large as 2064 Dwords. Thus, the CRC places a limit on the maximum size of a FIS. The limit is defined to be 2049 Dwords for SATA [2].

3.2.3 Scrambling

As stated in section 3.2, one of the functions that the link layer performs is EMI reduction. The CONT primitive does this for primitives, but it is also done for FISes. The contents of a frame, including the CRC, are scrambled before being sent. To do this, the data is XORed (a bitwise exclusive OR operation) with a pseudo-random number generator. Specifically, the PRNG used is a Galois Linear Feedback Shift Register (LFSR) [6].

At the start of each frame, the scrambler is reset. The receiver, using the same Galois LFSR, can then descramble the data by again XORing the data with the output of the scrambler. Primitives are never scrambled, even those sent in the middle of a frame (such as HOLD and ALIGN).

3.3 Transport Layer

The transport layer is responsible for constructing, delivering, and receiving Frame Information Structures. It defines the format of each FIS and the valid sequence of FISes that can exchanged.

The first byte of each FIS defines the type. The second byte contains type-dependent control fields. The following table lists some of the types of FISes that are defined, and the value of their type field.

FIS Type	Type Value
Register – Host to Device	0x27
Register – Device to Host	0x34
Data	0x46
DMA Activate	0x39

Table 4: FIS Types

A number of other FIS types are defined, but they are not implemented in this work. For more details on other FIS types, see [2] or [6].

The Register FIS types are used to transfer the contents of the shadow registers to the device, and the device registers back to the host. These registers mirror those used for PATA, and are the means by which commands are triggered. Some of the relevant fields are the Command field, which holds the PATA command to be executed; the addressing fields; and the sector count fields. A sector is 512 bytes [10].

The Data FIS is a very simple FIS. After the type field, the remainder of the first Dword is reserved. Following that is the actual data to be delivered. The maximum length of the data for a single FIS is 8KB. This is to ensure that the CRC is capable of checking the data.

The DMA Activate FIS is sent by the device to indicate that it is ready to receive data. After a write request has been made, the disk may need to prepare itself before it can receive data. For example, it may need to flush its buffer or move the head to the correct location. It is a very short FIS, consisting only of a single Dword. It contains the FIS type and the rest of the bits are reserved.

For read and write operations, the sequence of valid FISes is fairly simple. To perform a read, the host sends a Register – Host to Device (H2D Register) FIS to the disk with the PATA read command in the Command field. It then waits to receive one or more Data FISes (depending on the length of the operation) from the disk. After that, the device will send a D2H Register FIS to indicate its status.

Write operations are fairly similar. The host again sends an H2D Register FIS to the disk, but now with the PATA write command. It then awaits a DMA Activate FIS, indicating that the disk is ready. It then sends a Data FIS. If the operation is larger than 8KB, the host must wait for a new DMA Activate before sending each Data FIS. After the operation is complete, the device will again send a D2H Register FIS with status information.

3.4 SATA Conclusion

Overall, SATA is a very suitable protocol for the NASA SWOT mission. It allows for high speed storage compatible with almost any hard drive or SSD available on the market. Also, it is a very robust protocol, making it suitable for use in space. Each and every frame has a CRC to protect against bit errors. Low-voltage differential signaling adds noise immunity and decreases power consumed. There are also numerous methods employed to reduce EMI.

This section talked about SATA in general. Later sections will discuss the implementation of SATA on the Virtex-4. Note that the Command Layer was not discussed in detail. This is because, for the basic Read and Write operations that have been implemented, the Command Layer protocol is quite simple and consists mainly of transferring data to/from the Transport layer and checking to see if the operations have completed. For more details about the Command Layer, see [2] or [6].

CHAPTER 4

PREVIOUS WORK

4.1 XAPP 716

XAPP 716 is a Xilinx application note that describes an embedded SATA system for the ML405 evaluation board [7]. This system, referred to as the Embedded SATA storage reference system (ESS), was released in October 2006. The Application note contains details of the system and usage instructions, as well as some files for implementation and testing.

The ESS design makes use of the Virtex-4's embedded PowerPC processor, running a lightweight distribution of Linux. The design also includes an Ethernet interface, DDR memory, a memory controller, and a SATA hard drive interface. The core was tested with five different hard drives to ensure compatibility [7]. Included with the application note is a demonstration bitstream for the ML405 that will run for 10 minutes.

Unfortunately, the ESS design makes use of a proprietary SATA controller IP core, developed by ASICS World Services, which has a significant licensing cost [8]. This core handles the link layer and above of the SATA protocol. Without purchasing the license, this SATA core cannot be used in other designs.

However, the application note does include Verilog files and Constraints for some of the physical layer modules. Of note are the MGT initialization module, the OOB controller, and the DSOTDM (Dynamic SATA OOB Threshold Detector Module).

The DSOTDM is a solution to a problem in the silicon of the Virtex-4. The RocketIO parameter that determines the threshold for an out-of-band signal is named RXCDRLOS. However, this parameter does not have the accuracy intended, and a value that corresponds to one voltage level on one MGT may correspond to a different value on another MGT [7]. Thus, the correct value for this parameter must be found for each MGT and for each Virtex-4.

The DSOTDM solves this problem by changing the value of RXCDRLOS using the MGT's Dynamic Reconfiguration Port (DRP) before starting link initialization. By trying a range of values, the module can find a suitable value for the parameter. This module relies on the fact that a drive must always respond to a COMRESET OOB primitive with a COMINIT primitive. For each value to test, the module sends a COMRESET and checks for a response from the disk. A response indicates that this value of RXCDRLOS is valid.

The ESS's OOB controller includes a significant innovation. The Virtex-4 is unable to lock to the incoming data stream in the time specified by the SATA protocol. However, the OOB controller continues with the OOB sequence anyway, as if it is locked. This is possible because later in the start-up sequence, the device will wait for a long time for the host to respond. It is at this point that the host can wait for the lock to be acquired. Essentially the host is tricking the device; it indicates that it has acquired a lock by moving to the next step, but actually acquires the lock much later.

The ESS's MGT Initialization module is not used, because the newer Xilinx tools now include the RocketIO wizard, which generates modules to perform the required initialization procedure [9].

While the provided Verilog code is not used (in part because some files are missing), XAPP 716 has been a very useful resource for this work. The ESS reference design shows how to overcome some of the challenges associated with doing SATA on a Virtex-4. And perhaps even more importantly, it shows that it is possible.

4.2 The UNC Core

An open-source SATA core was created at the University of North Carolina at Charlotte by Ashwin Mendon, Bin Huang, and Ron Sass [10]. This core targets the Virtex-6, specifically the ML605 board. As such, the physical layer is quite different, since the high speed transceivers on the Virtex 4 and the Virtex 6 differ substantially.

The UNC Core implements all layers of the SATA protocol, and combines the Link Layer and Transport Layer into one module. The physical layer includes an instantiation of the transceiver and an OOB sequence controller. The Application Layer is a simple, FIFO-like interface that allows other hardware modules to easily read and write data.

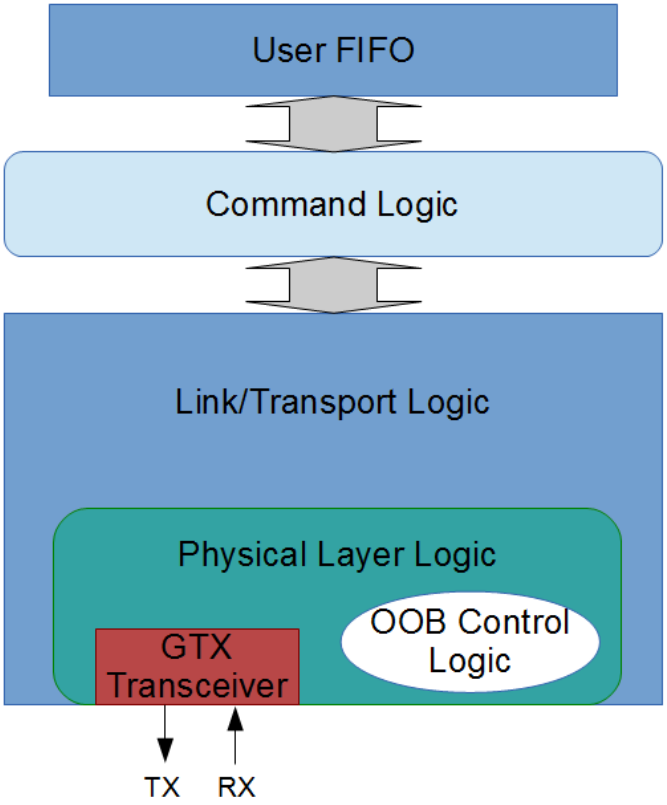


Figure 9: UNC Core Architecture

There are some significant differences between the RocketIO MGT present on the Virtex-4 and the GTX transceivers found on the Virtex-6. The most notable is support for OOB primitives. The RocketIO includes a signal that simply indicates the presence or absence of a signal; the GTX contains detectors for the actual primitives. It also includes generators for transmitting the OOB primitives. Another significant difference is the lack of a need for an initialization module. The GTX transceivers can also acquire a lock to the incoming data much more quickly and do not have problems with the OOB detection threshold [11].

The UNC Core implements the DMA Read and DMA Write commands. At the highest layer, the interface is presented as a simple FIFO. It does not support the PIO mode of SATA (a legacy mode added for ATA backwards-compatibility) or other commands and features, such as power management. However, reading and writing data is the primary purpose of the SATA interface, and other commands can be added if needed.

The link layer and transport layers of the protocol are combined in this implementation into a single module. This module contains submodules for performing scrambling, descrambling, and generating the CRC. It detects and generates link layer primitives. There are three state machines: one for the receive datapath, one for the transmit datapath, and one to control the sequence of FISes (referred to as the Master FSM).

The upper layers of the UNC core were used in this work and thus were pivotal for completion of the design. A new physical layer was created, but the interface to the link layer was designed to match that used by the original physical layer. Also, the layers were reorganized to make debugging easier. However, most of the logic remains intact, and thus we would like to take this opportunity to thank the developers of the UNC Core for their important contribution to the open-source community and to this work.

CHAPTER 5

THE VIRTEX-4

Virtex-4 refers to a series of devices in the Xilinx Virtex family. Virtex-4 devices contain a number of embedded, specialized blocks, including PowerPC processors, Ethernet MACs, Digital Signal Processing Slices, and blocks for high-speed clock management. And they also have, of special interest to this work, high-speed serial transceivers. Virtex-4 devices are created using a 90nm process and are 40% faster than Xilinx's older devices [13].

The board used for development in this work is an ML405 evaluation board. This board contains a Virtex-4 XCV4FX20 device. This device contains 19,224 logic cells and 1224Kb of block RAM. It also contains 8 RocketIO transceivers [13].

The DAS board is designed to use a radiation-hardened device from the Virtex-4QV family. The board actually holds two FPGAs—one for data processing and the other to handle the PCI interface. The Data FPGA is a Virtex-4 XC4VFX140. This device has 142,128 logic cells and 6,768 Kb of block RAM [13]. As can be noted, this is significantly larger than the device on the ML405. Thus, if a SATA design can be implemented on the ML405 without using a significant percentage of the resources, then it is very likely that the design will fit on the DAS Board, even alongside the other processing logic.

As stated, the Virtex-4 devices contain a number of high speed serial transceivers referred to as RocketIO MGTs (multi-gigabit transceivers). These transceivers can handle line rates from 622 Mb/s to 6.5 Gb/s [12]. Recall that the line rate of SATA I is 1.5 Gb/s, which falls into this range. The MGTs also have their own specialized clocking resources on the FPGA.

5.1 RocketIO

The RocketIO transceivers, being central to this work, will be discussed in some detail in this section. Further information can be found in the "Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide" [12]. The MGTs have many other features, not relevant to SATA, which will not be discussed here.

RocketIO MGTs are built-in hard blocks on the FPGA for handling high speed serial data. Each MGT has its own set of pins on the FPGA, which can be routed to a variety of different connectors. The ML405 board has two SATA connectors corresponding to two MGTs in the FPGA. Thus, on the ML405, two SATA devices can be used simultaneously, or they can be connected to each other with a special SATA loopback cable. This is very useful for testing. The DAS board has a single SATA connector, and the other MGT in that tile is connected to a small form-factor pluggable (SFP) connector.

The MGTs also have their own clocking resources. Another built-in block in the Virtex-4 is the GT11CLK module. Each of these modules has their own clocking pins on the FPGA. On both the ML405 and the DAS board, this is a 150MHz, low-jitter clock.

This clock speed allows the transceiver to run at SATA I speeds. SATA II speeds require a 300MHz reference clock (see Table 2-5 in the User Guide [12]). MGTs in the FPGA are organized into *tiles*, where each tile contains two MGTs. These MGTs share clocking resources. Thus, a single 150MHz oscillator can run both of the SATA MGTs on the ML405, as both MGTs share a tile.

It may at first seem wasteful that a separate clock is needed just to run the MGTs. However, the MGTs also provide clocks that can be used for FPGA logic. One is the TX clock, which is derived from the 150MHz oscillator on the board. The other is the RX clock, which is recovered from the incoming data stream using a PLL. Internal clock dividers in the RocketIO make these clocks the right speed for the interface width being used.

RocketIO MGTs are logically separated into a Physical Media Attachment (PMA) and a Physical Coding Sublayer (PCS). The PMA handles the actual serial interface, and the PCS is between the PMA and the FPGA fabric. The PMA contains the Serializer/Deserializer (SERDES) blocks to serialize and deserialize the data coming from and going to the FPGA fabric. It also handles clock generation and recovery. For SATA, the PLL's oscillator runs at 3000MHz, and this is divided down to a 750Mhz clock. The SERDES uses both edges of this clock to achieve the line rate of 1.5Gb/s. The PCS handles the clock domain crossing from the FPGA logic to the PMA. It also includes TX and RX buffers.

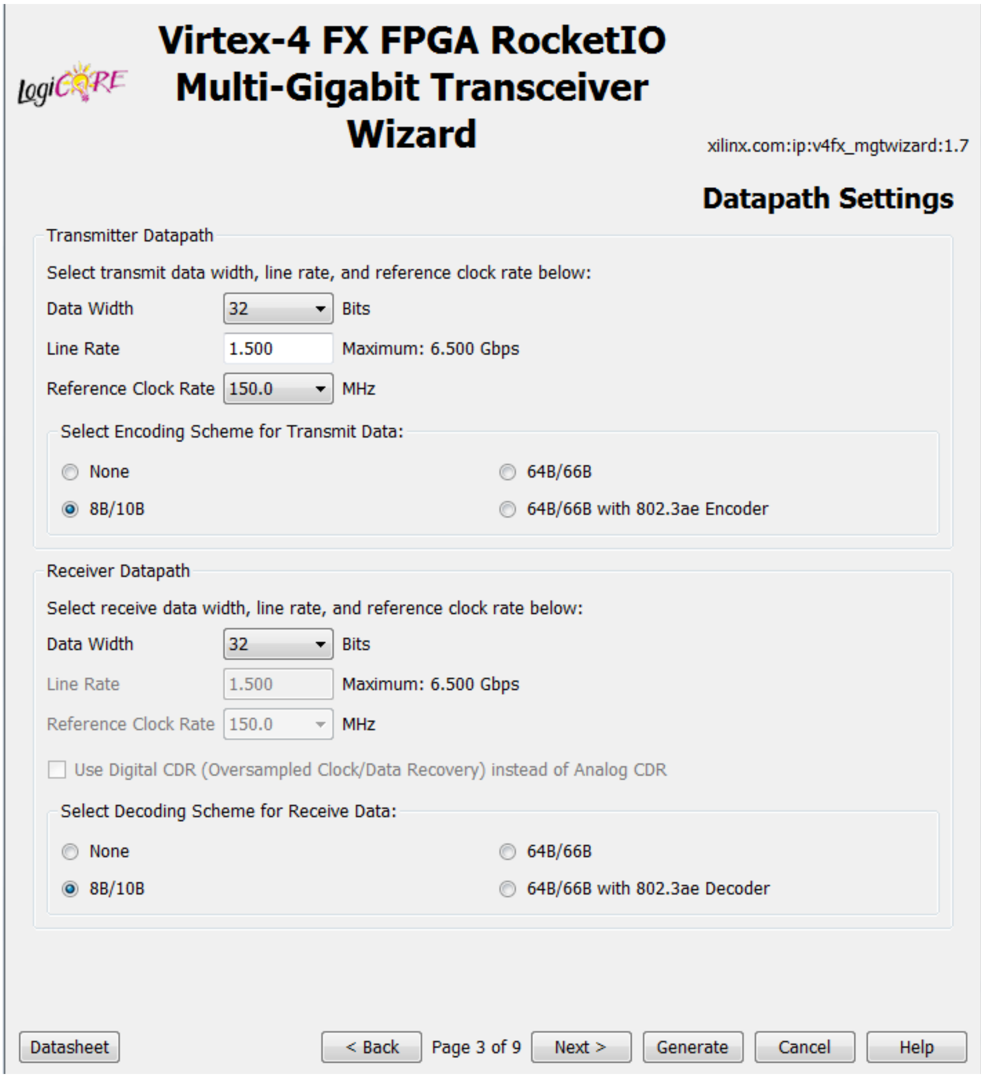
The RocketIO also includes a built-in 8b/10b encoder and decoder block. If selected for use, the bytes are encoded in the PCS before being placed in the buffer or being passed to the PMA. Another interesting feature in the RocketIO is the built-in CRC generator and checker. Unfortunately, this block cannot be used for SATA. That is, the CRC is scrambled along with the rest of the frame, and thus the RocketIO would need a built-in scrambler as well for this to be useful for SATA.

Another important feature of the RocketIO is its out-of-band signaling support. The MGT provides two ports to handle sending and receiving OOB signals. One port indicates whether or not a signal is detected on the line, and the other is used to send an OOB signal by driving the pins to a common voltage. Also, both of these signals can be used regardless of whether the receiver and transmitter are locked. This is important for link initialization.

The RocketIO also includes a loopback testing feature. In addition to testing using a loopback SATA cable on both the ports, this feature allows for testing using a single MGT. Data that is sent on the TX datapath is returned on the RX datapath, without going out over the cable. Furthermore, a second loopback mode allows for cutting out the PMA entirely to help isolate problems.

Virtex-4 RocketIO MGTs also include a Dynamic Reconfiguration Port (DRP). This allows for many of the MGT attributes to be changed at runtime. One use of this is to find the correct OOB voltage threshold for the receiver, as was done in XAPP 716.

A useful tool for creating the high-speed serial design is the RocketIO Wizard (a screenshot of the Wizard is shown in Figure 10). The wizard is a piece of software included in the Xilinx ISE tool kit. It contains a number of built-in protocol files that set the very numerous RocketIO parameters. For example, by selecting the SATA I protocol file and changing some settings, the wizard will determine the correct values for things like the clock dividers and electrical characteristics of the PMA. The wizard is actually necessary for developing with RocketIO, since the correct usages of many of the attributes are not documented anywhere in the User Guide. It also generates initialization modules and wrappers for the MGT blocks. These wrappers hide some of the unused ports to make working with the MGT easier.



Virtex-4 FX FPGA RocketIO
Multi-Gigabit Transceiver
Wizard

xilinx.com:ip:v4fx_mgtwizard:1.7

Datapath Settings

Transmitter Datapath

Select transmit data width, line rate, and reference clock rate below:

Data Width: 32 Bits

Line Rate: 1.500 Maximum: 6.500 Gbps

Reference Clock Rate: 150.0 MHz

Select Encoding Scheme for Transmit Data:

○ None
● 8B/10B
○ 64B/66B
○ 64B/66B with 802.3ae Encoder

Receiver Datapath

Select receive data width, line rate, and reference clock rate below:

Data Width: 32 Bits

Line Rate: 1.500 Maximum: 6.500 Gbps

Reference Clock Rate: 150.0 MHz

□ Use Digital CDR (Oversampled Clock/Data Recovery) instead of Analog CDR

Select Decoding Scheme for Receive Data:

○ None
● 8B/10B
○ 64B/66B
○ 64B/66B with 802.3ae Decoder

Figure 10: RocketIO Wizard Screenshot

The following table lists some of the ports and attributes that are particularly important for SATA. A complete list of the ports and their functions is available in the User Guide [12].

Port/Attribute	Description
RXN / RXP	The differential receiver pins.
TXN / TXP	The differential transmitter pins.

Port/Attribute	Description
RXLOCK	This port indicates whether or not the PLL has acquired a lock to the incoming datastream.
RXRECCLK1	The recovered and divided clock from the datastream. It uses the local oscillator as a reference.
TXOUTCLK1	A clock derived from the local 150 MHz oscillator.
RXSIGDET	This signal indicates the detection of an OOB signal.
TXENOOB	This port is used to send OOB signals.
RXCHARISK	This indicates that the received character, after passing through the 8b/10b decoder, is a K character.
TXCHARISK	This indicates to the 8b/10b encoder that the corresponding character should be encoded as a K character.
ENMCOMMAALIGN / ENPCOMMAALIGN	These ports enables the received data to be re-aligned to commas. For SATA, these should always be 1.
RXDATA	The received data, after being decoded.
TXDATA	The data to be sent, before encoding.
RXUSRCLK2	This is an input port that handles the clocking of data between the MGT and the FPGA fabric.
TXUSRCLK2	This handles clocking for the transmit data.
LOOPBACK	This port sets the internal loopback mode for the MGT. This is very useful for testing.
REFCLK1	This input is for the local reference clock. The 150MHz oscillator drives a GT11CLK module, which creates this signal.
RXCDRLOS	This attribute determines the voltage threshold for the boundary between an OOB signal and an in-band signal.
ALIGN_COMMA_WORD	Determines which byte boundary commas align to. A value of 1 means that commas align to any byte, while a value of 4 means that commas will be Dword-aligned.
MCOMMA_32B_VALUE / PCOMMA_32B_VALUE	These attributes define the comma value for alignment for positive and negative parity. For SATA, these values are set to K28.5.
MCOMMA_DETECT / PCOMMA_DETECT	If set to true, a flag will be raised whenever a comma is detected.

Port/Attribute	Description
RX_BUFFER_USE	This attribute can be used to bypass the receiver's buffer.
TX_BUFFER_USE	Whether or not to bypass the transmit buffer.

Table 5: Important RocketIO Ports and Attributes

As stated, this is only a partial list, and the RocketIO MGT contains many more ports and attributes. Most of these are unchanged from the values given to them from the RocketIO Wizard.

The MGT has a variable-width interface. A width of 1 byte, 2 bytes, 4 bytes, or 8 bytes can be selected. This selection determines the bus width of RXDATA and TXDATA, as well as the CHARISK ports (each byte could be a K character). Internal dividers then modify the clock speeds (RXRECCLK1 and TXOUTCLK1) to be appropriate for that data width. For example, for a 1-byte datapath width, the clocks available at the FPGA fabric would be 150 MHz. This is because the line rate is 1.5Gb/s, but each byte is encoded to ten bits instead of eight (See 3.1.2 for details of 8b/10b encoding). So 1.5Gb/s divided by 10 is 150MB/s. The RocketIO samples new data from the FPGA fabric at every clock tick, so the clock speed must be 150Mhz. By a similar calculation, we find that a 4-byte datapath width yields a clock speed of 37.5MHz.

These output clocks can be used to drive the entire design by buffering them on the FPGA's global clock net. That way, everything can be in synchronization with the MGT. The USRCLK2 ports are inputs to the MGT, and the same clock used to drive the logic should be connected to these ports. Thus, the clocks are generated from within the

MGT, buffered globally and used to drive the logic, and then fed back to keep the MGT/FPGA interface synchronized.

CHAPTER 6

DESIGN DETAILS

We will now begin a discussion of the design of the SATA core itself. In short, this design uses a newly-created physical layer alongside the upper layers of the UNC Core. The upper layers have been augmented with new error detection and correction features to improve reliability. This new core has been extensively tested on both the ML405 board and the DAS board. Newly-created debugging modules were used to assist the design and testing of the core.

At the top level, the SATA core presents a simple, FIFO-like interface that other modules could use to store and retrieve data. The complicated workings of the SATA protocol are abstracted away to make the core easy to use.

The Physical Layer instantiates two RocketIO MGTs, one for each SATA connector on the board. This allows for loopback debugging. Also instantiated is the CLK module and the initialization modules. The actual SATA logic drives one of the connectors, and the other is tied to a simple state machine created for debugging. This state machine sends a repetitive pattern of link layer primitives that exercise most states of the SATA core.

It would also be possible to instantiate two SATA cores simultaneously, with each driving one MGT. This would allow for simultaneous communication with two separate disks.

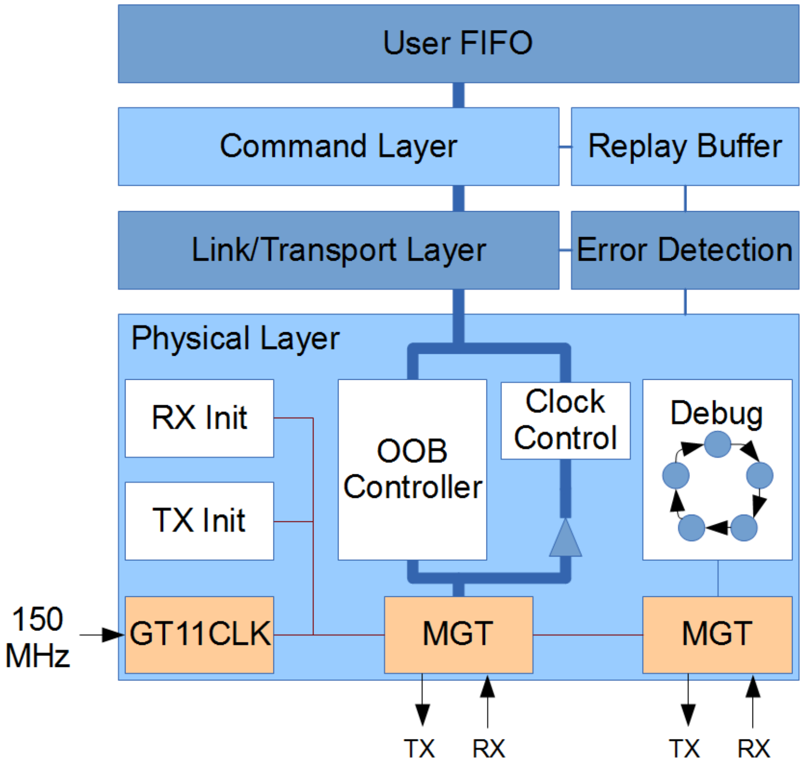


Figure 11: SATA Core Block Diagram

As stated earlier, the core must be able to respond to a HOLD within 20 Dwords being sent. To meet this timing constraint, the RocketIO MGTs are instantiated with the TX and RX buffers bypassed. The RocketIO wizard creates the initialization modules that are necessary to use the MGTs in this mode. With loopback cable testing, we find that the time between sending a HOLD and receiving it is 16 Dword clocks (this could vary with the length of the cable). Thus, the bufferless mode can meet the HOLD time constraint.

For this design, we have selected a 32-bit datapath for the RocketIO. The reason for this is that all link layer primitives are 32-bits wide; thus, it makes transmitting and detecting the primitives easier if the interface has this width. Also, the UNC core uses a

32-bit datapath, and therefore using the same width makes integrating the physical layer easier.

6.1 OOB Sequence Controller

The OOB sequence contains the primary logic for the physical layer. It handles the initial handshaking sequence and ALIGN insertion.

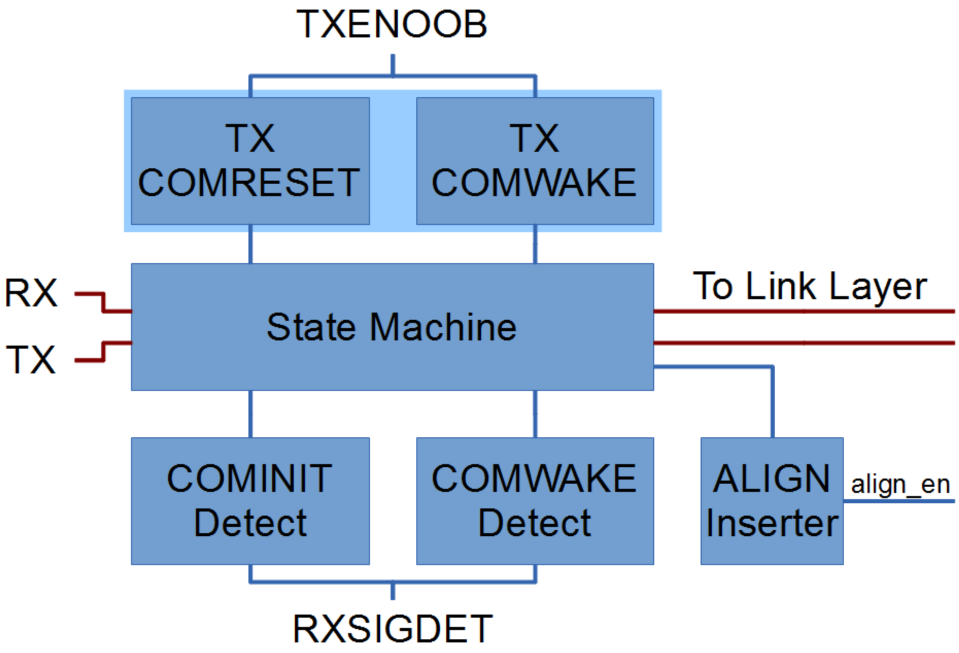
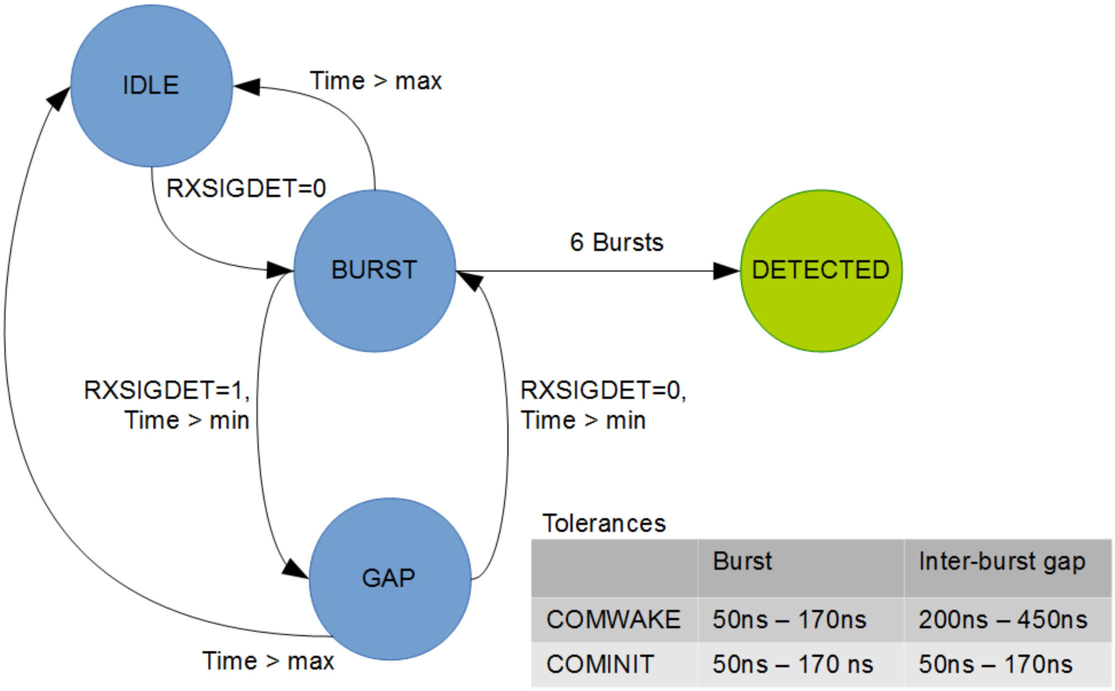


Figure 12: OOB Sequence Controller Block Diagram

As shown in Figure 12, the OOB controller consists of multiple smaller modules that handle the creation and detection of OOB primitives. The detection modules use a simple state machine to check the timing of OOB pulses on the RXSIGDET port. If the pulses match an OOB primitive within some tolerances, they assert a flag indicating detection.



Tolerances

	Burst	Inter-burst gap
COMWAKE	50ns – 170ns	200ns – 450ns
COMINIT	50ns – 170 ns	50ns – 170ns

Figure 13: OOB Primitive Detection State Machine

The transmission modules work in a similar manner. They use a state machine to time the sending of OOB pulses using the TXENOOB port of the MGT. The two transmission modules, tx_comreset and tx_comwake, are wrapped in another module that prevents them from interfering with each other (for example, by trying to send both types of OOB primitive at once). It also controls TXENOOB in between the sending of primitives. Per the specification, TXENOOB should be high between the primitives, but low once the host starts sending the dial tone pattern.

The OOB sequence has 8 states, which closely follow the initialization procedure discussed previously. Note that the state machine flow is cyclical; once it starts, the controller will repeatedly attempt to initialize communications until it is successful or reset.

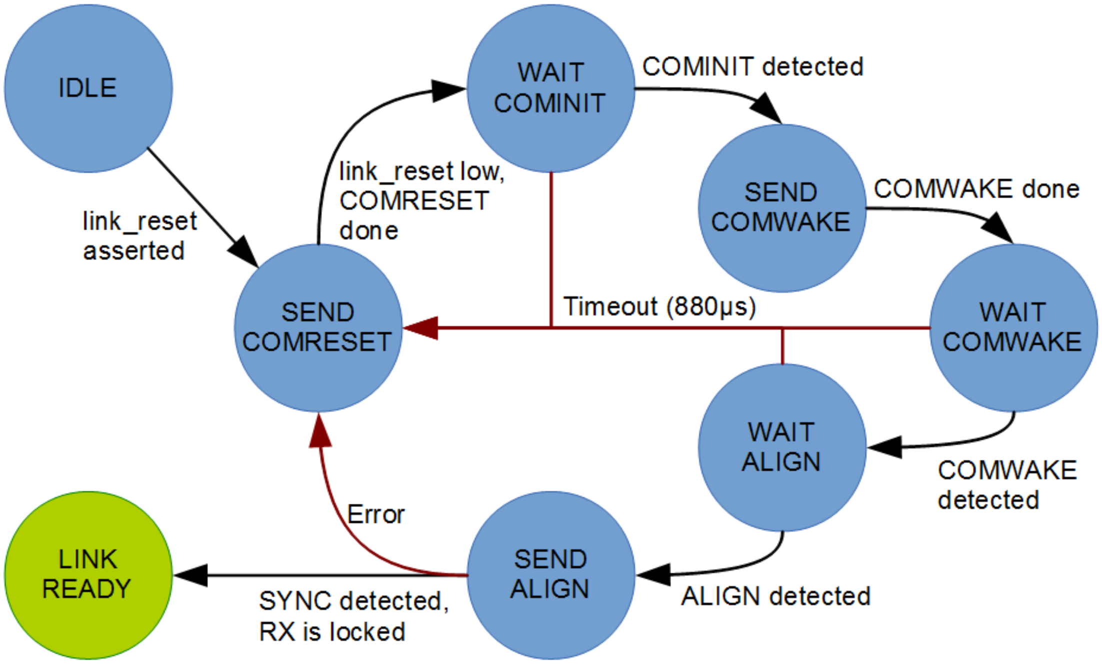


Figure 14: OOB State Machine Diagram

The first state is the idle state. Here, the controller simply waits and acknowledges any flags from the OOB primitive detectors. The reason for this is because many drives will send COMINIT periodically, even when not prompted by COMRESET, to announce that they are connected. However, this OOB controller waits for instruction from the rest of the core before moving forward.

The core triggers the start of the OOB sequence by asserting the link_reset signal. The sequence controller then moves to the send_comreset state. This state asserts the comreset_send signal, which triggers the transmission module. It then waits for the transmission of COMRESET to complete and moves to the wait_cominit state.

The wait_cominit state, as expected, waits for a COMINIT to be detected. It then acknowledges the detection module and moves on to the next state. However, if no COMINIT is detected after 880us, it moves back to the send_comreset state, restarting the sequence.

The next state is send_comwake. It triggers the tx_comwake module, waits for it to finish, and then moves to the next state.

Wait_comwake is almost identical to wait_cominit, except, of course, that it looks for COMWAKE. It also has a timeout that restarts the sequence. One difference is that, when COMWAKE is detected, it turns OOB signaling off, since the rest of the sequence uses in-band signals.

The wait_align state is the first that uses in-band signals. In this state, the controller sends the "dial tone" pattern of alternating ones and zeros to the device. Meanwhile, it waits for an ALIGN primitive. Since the host only operates at SATA I speeds, we do not need to worry about speed negotiation. The device may try sending at a faster rate at first, but this will appear to the core as garbage. Once we are able to positively identify an ALIGN primitive, the controller moves to send_align.

In this state, the controller sends ALIGN to the device to acknowledge this speed. The device will respond with SYNC and expects that the host will then send SYNC as well. However, this state does not only wait for SYNC, it waits for rx_lock as well. This signal indicates whether or not a lock to the incoming data stream has been acquired. The controller sends ALIGN until the lock is acquired and SYNC primitives are detected. The device will send repeated SYNCs for a long enough time that the lock can be obtained. If an OOB signal is detected, an error has occurred and the drive has returned to the beginning of the sequence. If this happens, the controller moves back to send_comreset to restart the process.

Once SYNC is detected and the lock is ready, the controller moves to the link_ready state. It sends SYNC back to the disk and indicates to the rest of the core that the link is up.

In this implementation, the OOB controller also handles the necessary insertion of ALIGN primitives every 256 DWords once the link is ready. A counter keeps track of the number of DWords sent and asserts align_en for 2 cycles out of every 256. This signal is sent to the upper layers of the core one cycle in advance so that processing can be paused without loss of data. Otherwise, an actual Dword may be "overwritten" by an ALIGN.

6.2 Interface

This design uses the upper layers of the UNC core to handle the Link Layer, Transport Layer, and Command Layer of the SATA protocol. As stated, the UNC core was originally designed for a Virtex-6 FPGA, and thus is not readily compatible with the Virtex-4. However, the layered design of SATA makes integrating the new physical layer with the upper layers rather easy. As long as the interface between the physical layer and the link layer remains the same, the upper layers' logic will still work.

Therefore, the new physical layer has many of the same ports as the original UNC physical layer. The MGT-specific ports are different, but the ports for the interface between the physical layer and the link layer are the same. What follows is a description of the physical layer's ports.

Port(s)	Description
UPPER_MGTCLK_PAD_N_IN, UPPER_MGTCLK_PAD_P_IN	These are the outputs of the 150MHz oscillator that drives the GT11CLK module.
CLK_100_IN	A separate, 100MHz clock. The UNC core, which targeted the ML605 board, used a 150MHz clock. The DAS board has a 60MHz oscillator, so this will need to be changed when porting the design.
TX_SYSTEM_RESET_IN, RX_SYSTEM_RESET_IN	Resets for the TX and RX datapaths.
MGT_RXLOCK_OUT, MGT_TXLOCK_OUT	Outputs that indicate whether the RocketIO's PLL has locked. For TXLOCK, which uses the local 150MHz oscillator, this lock occurs almost immediately. RXLOCK must wait for part of the OOB startup sequence, as described above. Before locking to the data, this signal will toggle on and off. Both of these ports are useful for debugging.
RX1N_IN, RX1P_IN; TX1N_OUT, TX1P_OUT	The differential receiver and transmitter pins for SATA.
tx_datain	A 32-bit input from the link layer, which is the data to be sent. Data is transmitted every cycle.
tx_charisk_in	A signal indicating whether the last byte of tx_datain should be encoded as a K character. This is passed to the MGT.
LINKUP, LINKUP_led	Outputs that indicate completion of the OOB start-up sequence. This signal triggers activity on the upper layers.

Port(s)	Description
align_en_out	Indicates that an align is being inserted, as per the specification. This allows the link layer to pause its processing.
sata_user_clk	The divided clock that is output from the MGT. This clock drives all the logic in the upper layers.
rx_dataout	32-bit data that has been received.
rx_charisk_out	A 4-bit signal indicating if any of the bytes in rx_dataout were received as K-characters.
rxelecidle_out	RXSIGDET; whether or not an in-band signal is present.
oob_state	The current state of the OOB sequence controller state machine. Used only for debugging, this signal has no bearing on the logic of the upper layers.
force_ready	Used for loopback debugging, this signal bypasses the OOB start-up sequence.
sata_phy_ila_control, sata_loopback_ila_control	These ports are for the Chipscope ILA debugging cores. Debugging will discussed shortly.

Table 6: Physical Layer Interface

6.3 Clocking

The physical layer also handles clock management. As stated in section 5.1, the RocketIO MGTs have two output ports for clocking: RXRECCLK1, which is recovered from the incoming bitstream, and TXOUTCLK1, which is derived from the local oscillator. These clocks can be used to drive the SATA hardware.

As it turns out, choice of clocking configuration is very important for reliability. One possibility would be to use both clocks, one to drive the receive datapath and the other for the transmit datapath. However, the UNC core runs on a single clock, so implementing this configuration would necessitate significant changes in the upper layers. Thus, we need to use a single clock to drive the SATA logic.

The choice that seems correct at first would be to use RXRECCLK1, because of spread-spectrum clocking (SSC). As discussed in section 3.1.3, the RocketIO is able to receive spread-spectrum clocking signals but does not use it when transmitting. Since SSC only occurs downspread, this means that the receive clock will be slightly slower on average than the transmit clock. Using the slower clock would ensure that no received data is missed.

Unfortunately, we find that the SATA core does not operate reliably using this clocking configuration, as Figure 15 shows. The clock derived from the incoming bitstream is not stable enough to drive the core.

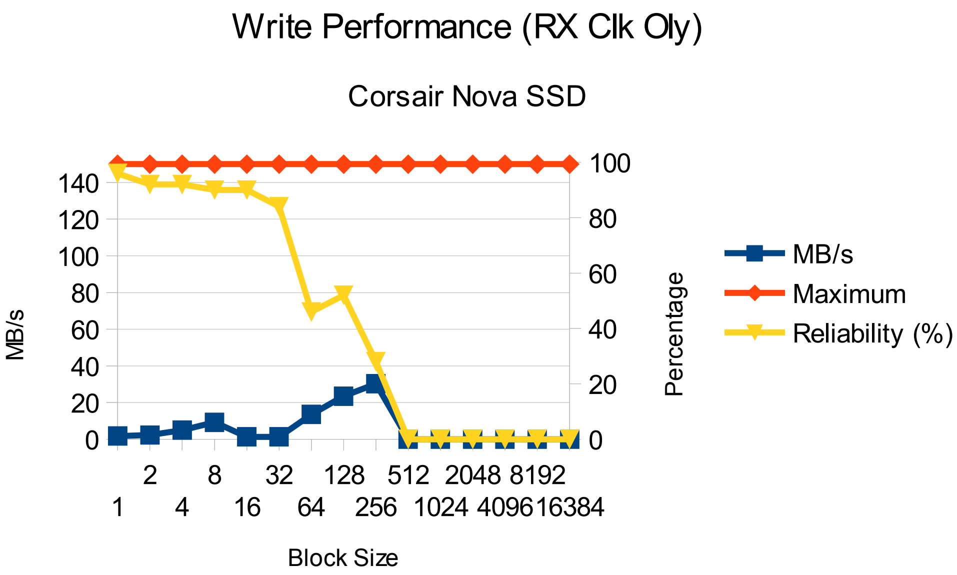


Figure 15: SATA Core Performance using RXRECCLK1

This data was generated by sending 500 sequential write operations of varying sizes. The number of clock cycles taken to complete the transfer is used to determine the average throughput (MB/s). Transfers that do not complete correctly are not counted for the throughput.

Here, reliability is measured as the percentage of SATA operations that terminate correctly. Note that this decreases as the block size of the write operation increases. This is because larger writes require sending more FISes (recall that the maximum size of a FIS is 8KB, or 16 sectors). Therefore, there are more chances for a FIS transfer to fail.

We would also expect that the throughput (MB/s) would increase with the block size, since there is less protocol overhead. Here, the throughput increases gradually to a maximum of 30MB/s, before reaching a point where no operations can complete correctly.

Since the performance of the core under this configuration is quite poor, other configurations were also tried. The next most obvious clocking scheme is to use TXOUTCLK1. The results of this test are shown in Figure 16.

Here, the reliability is much improved, but still dips below 99%. The throughput is significantly improved, especially for write sizes of 16 sectors (exactly one data FIS per operation).

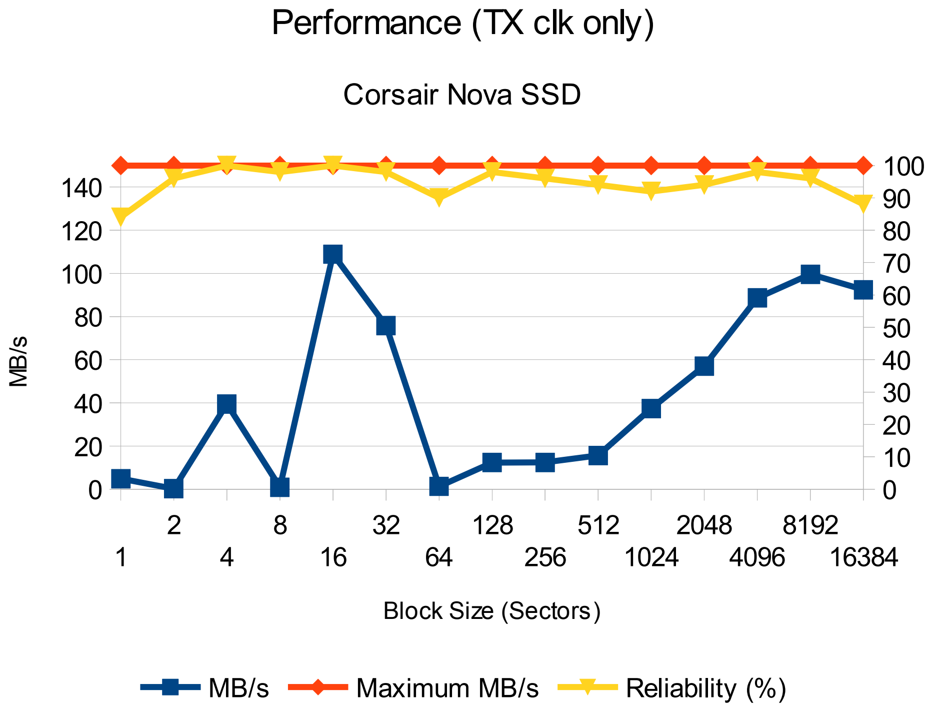


Figure 16: SATA Core Performance using TXOUTCLK1

However, as stated previously, the transmit clock runs slightly faster on average than the receive clock, so some incoming data is lost with this configuration. If part of a FIS (such as a DMA Activate) is lost, the transfer will not complete. This also means that read operations would likely be missing some data.

The solution is to use both TXOUTCLK1 and RXRECCLK1. Incoming data is written to a small clock-domain-crossing FIFO. Data in this FIFO is written using the receive clock and read using the transmit clock. In this way, we can use the stable TXOUTCLK1 to drive the logic of the core, while ensuring that no incoming data is lost.

Of course, this raises the question: what happens when this FIFO underflows? As stated, the transmit clock is faster on average than the receive clock, so data will be read slightly faster than it is written. A small module in the physical layer monitors for this underflow condition, and inserts an ALIGN primitive in the datastream when it occurs. ALIGN primitives are already present in the stream since the protocol requires that they be sent every 256 Dwords, and the Link Layer simply filters them out. These new ALIGN primitives are filtered as well, and thus this insertion does not have any negative impact on the operation of the core.

The performance of the core using this clocking scheme is shown in Figure 17.

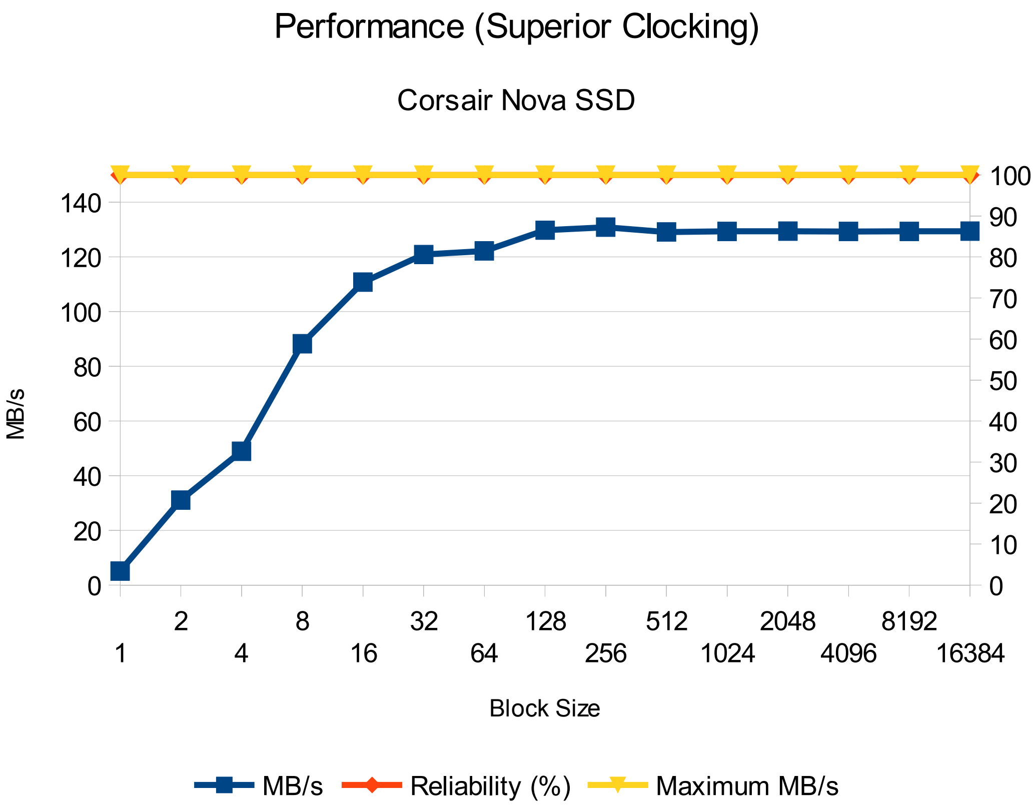


Figure 17: SATA Core Performance using the superior clocking scheme

With this configuration, the core has a reliability of greater than 99.99%. The throughput achieves a maximum of 130MB/s, close to the theoretical maximum of first generation SATA (150MB/s).

6.4 Error Detection and Recovery

Although most data write transfers will finish without problems, we found that a small number (less than 0.001%) would result in error. As written, the UNC core does not have any error detection features. Errors cause the core to pause (or “hang”) until it is reset. It is desirable for the core to recovery from these errors gracefully, preferably in a way that is invisible to the rest of the design.

To facilitate this, a replay buffer has been added to the Command Layer. The replay buffer mirrors the most recently sent Data FIS. When an error occurs, a flag is raised. If the error occurred on a Register Host to Device FIS, the Command Layer simply sends it again. If the error occurred on a Data FIS, then the Command Layer sends the data from the replay buffer instead of the User FIFO. Thus, to the rest of the hardware design, it appears as if there was no error; instead, it seems that the SATA core is operating more slowly.

A new handshaking signal has been added to the Link Layer that indicates successful transmission of a FIS. When this happens, the command layer flushes the replay buffer and sends the next FIS to the Link Layer. Of course, this new data is also mirrored in the replay buffer in case of an error. The replay buffer is 8KB in size, which is the maximum allowable size for a single FIS.

The simplest error and the easiest to recover from is a bit error. This occurs when the device reports a bad CRC or a parity error by sending the R_ERR primitive. The original UNC core would simply hang in this case, requiring a reset of the entire core. With the new error detection features, the Command Layer simply starts a new transfer using the replay buffer, as described above.

The Error Detection module monitors for other types of errors as well., but these are not as easy to recover from. One such error is a framing error, where a SOF or EOF primitive was received incorrectly, causing the host and device to be out of sync. To recover from this, the link must first be reset. A flag is raised at the Command Layer, indicating the error, and the Link Layer and Physical Layer are issued a reset. The Command Layer is not reset, so that the replay buffer can be used and no data is lost.

Unfortunately, resetting the link in this way results in a performance hit. The physical layer has to undergo the entire OOB sequence once again. From testing, this takes around 45000 SATA user clock cycles, or about 1.2ms.

As mentioned in section 3.2, there is a mechanism called the “SYNC Escape” that can be used to issue a soft reset for the link. Using this, the link can be returned to the idle state after a frame error in less than 10 cycles. Unfortunately, the disk takes a long time to respond after this operation; over 9,000,000 cycles pass before it will send R_RDY to receive any type of FIS. It also causes the disk to switch to a mode that does not use the

CONT primitive, which appears to be undocumented behavior. The hard reset operation is much better in comparison.

The error detection module can also detect a loss of link condition. This can happen when the disk enters an error state and stops communicating. The detection module can detect when this happens because the Physical Layer will report a “linkup” state, but the rxsigdet port will also be high, indicating no incoming signal. In this case, a hard reset is the only option.

The error detection and correction module adds robustness to the SATA core. Without it, the entire core would need to be reset in the case of any type of error. This would require an outside module to detect that the core is stuck, which typically means waiting for a significant amount of time. This could also lead to data loss. These new features make the SATA core much more reliable.

CHAPTER 7

TESTING METHODOLOGY

Testing has been very important at each stage of the design. Because of the high speed of the serial communication, and because the behavior of the disk cannot be controlled or changed, debugging the design has been rather difficult. There are many places where a bug could originate—the disk itself, the physical set-up, the MGT (recall the numerous parameters), or the design logic. Therefore isolating and resolving bugs can be arduous. This section will describe some of the methods and resources that were used for debugging and testing, as well as performance tests on the ML405 evaluation board.

7.1 Simulation

Simulation is the preferred method of debugging a hardware design. It allows the designer to have a complete view of all signals in the system. Also, the compilation time is significantly shorter, since the design does not need to be fully synthesized to the FPGA. To test a design, a simulation testbench is used to exercise some aspect of the design by driving the input signals.

Each module of the OOB sequence controller was simulated individually to check for the correct behavior. A simulation testbench was created that exercises the RXSIGDET signal in the same pattern as the COMINIT and COMWAKE primitives. The width of the pulses was also varied to check if the module could handle the protocol-specified tolerances.

The primitive transmission modules were tested similarly. The send_en signal was asserted and the resulting pulses on TXENOOB were checked to make sure that they were correct.

After each individual module was tested, the OOB sequence controller was simulated as a whole. A testbench containing the entire start-up sequence was created to exercise the controller. The simulation waveforms showed that the controller was successfully able to complete the sequence.

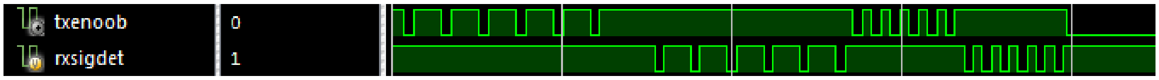


Figure 18: OOB Simulation Results

The above screenshot from the Xilinx ISim simulator shows the OOB sequence controller completing the first part of the sequence.

Unfortunately, no simulation testing was done beyond the OOB sequence. The entire SATA protocol is quite complicated, and there is no complete simulation model for hard drive behavior. While it would be possible to create a hard drive model, this would take a significant amount of time and it would be extremely difficult to verify correctness. And if the hard drive model did turn out to be incorrect, then much of the time spent designing and testing would have been a waste, since it was done based on the wrong assumptions. Therefore, further testing of the core uses an actual SATA hard drive, with the design being synthesized in hardware.

7.2 Chipscope

Chipscope is a set of tools used for debugging Xilinx hardware designs. Chipscope cores, including logic analyzers and virtual I/O ports, can be inserted into a design. These cores can capture and store internal signals in the logic. The signals can then be viewed using the Chipscope Pro Analyzer, a piece of software running on a PC that interfaces with the cores. The designer can set up trigger events to capture the data, and then view and save them on the PC.

Multiple Chipscope cores can be instantiated simultaneously in a design. This allows for the debugging of multiple modules and sub-modules without the need to re-synthesize the design. The signals that have been chosen to be captured are sampled at the clock rate, and then stored in Block RAM. Thus the number of samples is limited by the amount of available memory on the chip.

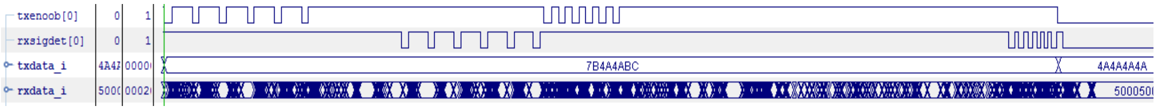


Figure 19: OOB Sequence Chipscope Screenshot

The Chipscope tools have been crucial for the work in this project. The integrated logic analyzer (ILA) cores give a clear view of what is happening internally. An ILA is instantiated in the module for each layer, adding debugging capability to each aspect of the design. Many problematic bugs have been discovered via Chipscope.

The different Loopback modes of the RocketIO MGT were also used alongside Chipscope for debugging. Since the internal MGT signals cannot be accessed, the different modes were very helpful in isolating problems and determining if the cause was physical, with the MGT parameters, or in the design logic.

7.3 SATA Event Logger

Although Chipscope has been extremely useful, it does have its limitations. As stated in the previous section, the number of samples that can be recorded is limited by the available memory of the device. Since SATA events are often separated by a significant amount of time, and since any operation will involve numerous frames being sent and received, this limitation makes debugging higher-level problems difficult. Also, although multiple ILAs can be used simultaneously, they cannot be set to trigger simultaneously. This means one cannot easily correlate events across different modules. For example, the physical layer ILA shows exactly what is being sent and received, but not the current state of the link layer state machine. That could be captured with the link layer ILA but would require a separate trigger. It is impossible to see what is happening in different ILAs at the same time.

To get a wider view of what happens in the core, we have created a new debugging tool. This tool, called the SATA event logger, utilizes the serial port to transfer debugging information to the PC. This way, the limited memory resources on the FPGA are not used.

As stated, SATA events are often separated in time, and since SATA runs at such a high speed, it is not possible to send all the data over the serial port. Instead, we only consider the SATA events. These events include sending and receiving FISes, responses from the disk such as R_OK or R_ERR, and successful read/write operations.

For the design of the event logger, the events are encoded as 8 ASCII characters, to make them readable in a terminal. A monitor checks for occurrences in the link layer module and pass the encoded event to the SATA event logger module. These events can come from multiple sources in the link layer module, so it is necessary to arbitrate between them. When a new event arrives, the event logger adds a timestamp and places the event in a FIFO. The timestamp is a simple counter that is wide enough to not overflow. The arbiter simply checks the timestamps of events coming from the FIFO and selects the earliest one. The timestamp is then ASCII-encoded, and passed to the serial interface module.

The SATA event logger has four FIFOs, so it can receive events from four sources. One source is a monitor that checks for new incoming primitives and also reports on the state of the RX lock. The other sources are the RX datapath, the TX datapath, and the Master FSM (which handles the sequence of FISes) of the Link Layer module.

Unfortunately, due to the vastly different speeds between SATA and the serial port, these FIFOs often overflow, resulting in incorrect debugging information. However, the individual FIFOs can be turned off or on as needed, as can the events to monitor. Despite these limitations, the SATA event logger proved very useful for solving the

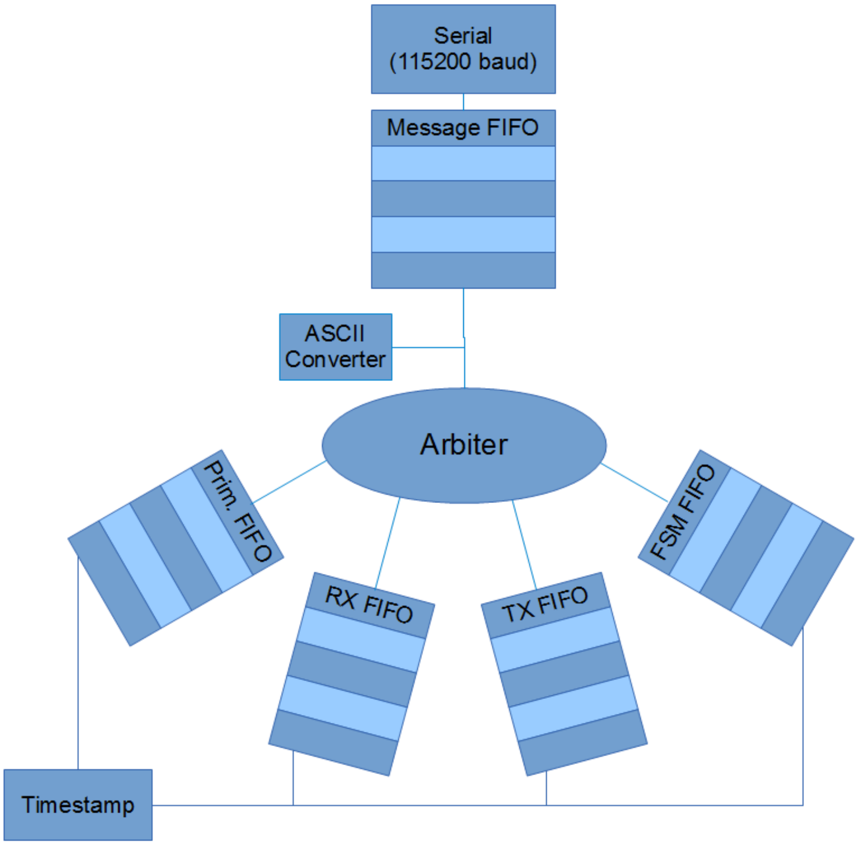


Figure 20: Design of the SATA Event Logger

clocking problem in the physical layer and for adding the error correction and recovery features.

7.4 SATA Performance

After solving the reliability problems, the SATA core was tested on the ML405 board, against three different hard drives. The makes and models of the hard drives used are shown in the table below.

Brand and Model	Type
Corsair Nova	SSD (Solid State Drive)
Seagate Momentus 5400.6	Spinning/Winchester
Western Digital Caviar SE WD1600JS	Spinning/Winchester

Table 7: Test Hard Drives

A Microblaze soft core processor is used to perform the tests. A small control module was created to interface the SATA core with the Microblaze. This allows the processor to set various parameters, such as sector count and address, and initiate data transfers. The control module also gives status information to the processor.

A simple software test application runs the tests and returns data to the console window over the serial port. This application issues 500 write commands of varying block sizes. After each, the number of cycles taken to complete the transfer is stored and used to calculate the average throughput in MB/s. A transfer is considered a failure if it takes more than 3 seconds to complete. In this case, the application resets the SATA core and continues.

The data written to the disk in the test is a simple counter fed into the user FIFO of the SATA core. The FIFO is always kept full so as to test the true maximum transfer rate.

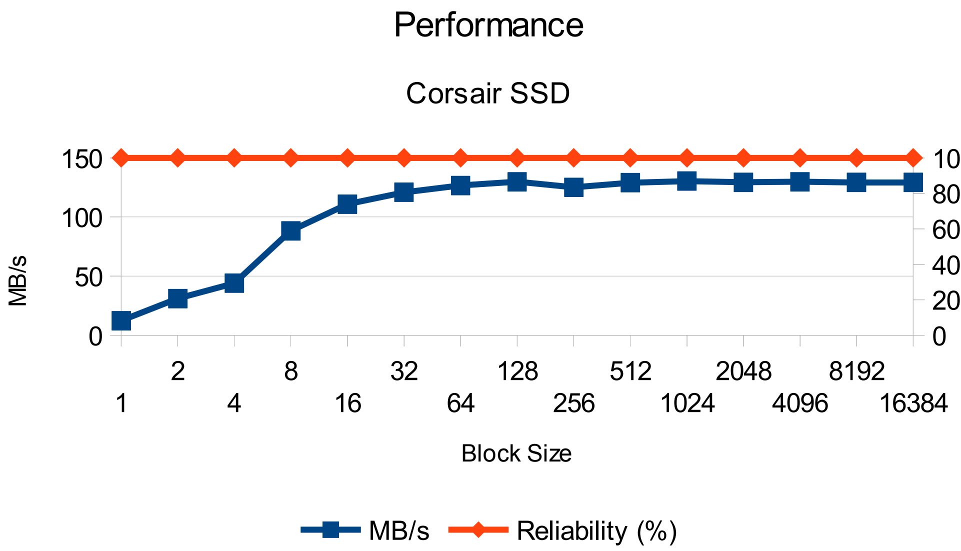


Figure 21: Corsair SSD Performance Test

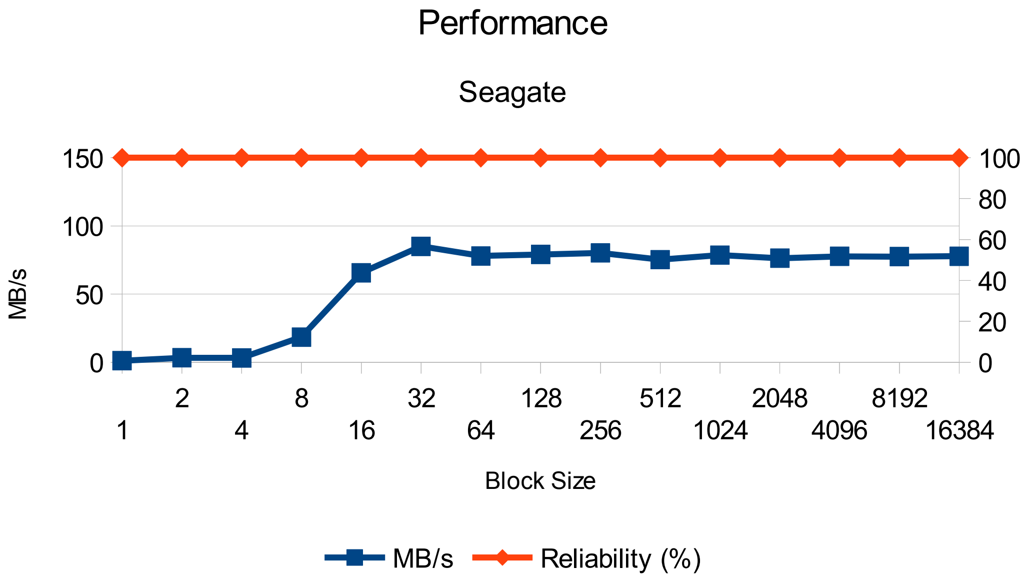


Figure 22: Seagate Performance Test

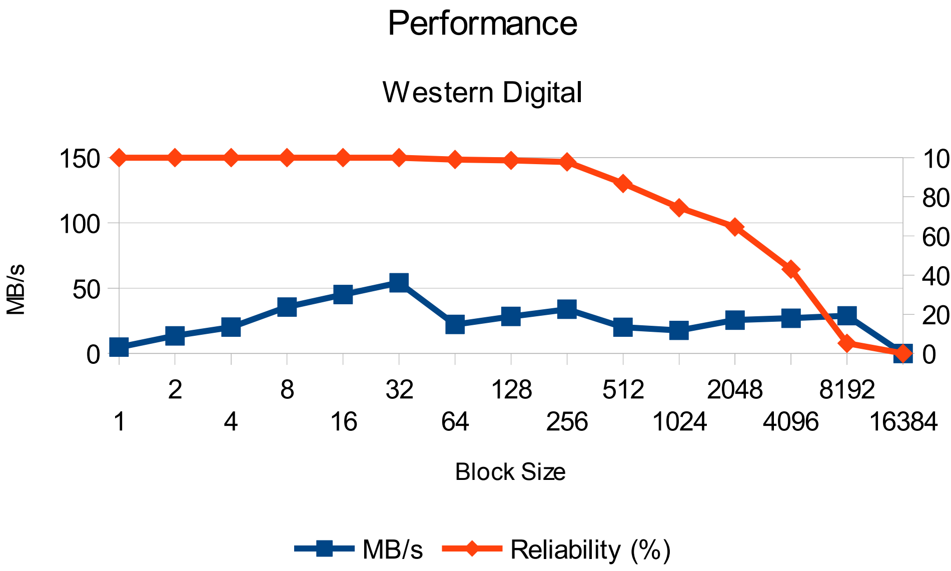


Figure 23: Western Digital Performance Test

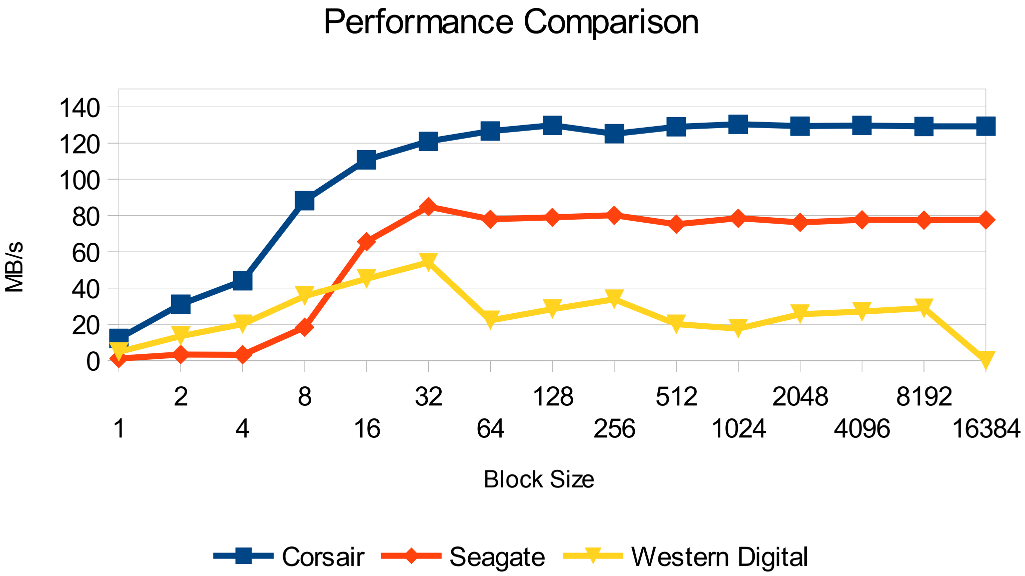


Figure 24: Hard Drive Performance Comparison

As can be seen in the above figures, the Seagate and the Corsair SSD have similarly-shaped performance curves. The Seagate does have a small spike at write sizes of 32 sectors (16KB), where it achieves its maximum performance of 85MB/s. The SSD has a maximum write speed of 130MB/s at block sizes of 1024 sectors (512 KB).

The Western Digital has the worst performance and reliability of the three disks tested. For write sizes of 32 sectors or less, it is reliable, but then the reliability starts to drop off. It achieves its maximum throughput of 54MB/s at block sizes of 32, so it would seem that it is optimized in some way for writes of this size.

The failures at larger write sizes are caused by the disk sending the HOLD primitive continuously for longer than 3 seconds. The protocol specifies that the host must reply HOLD_ACK, so there isn't anything that the core can do to correct this. At write sizes of 16384, this disk will never send a DMA Activate FIS to allow the transfer, so none of the operations can be completed. It would seem that the Western Digital hard drive was designed only with small writes in mind.

These performance tests show that the SATA core can operate correctly with a variety of disks. The Western Digital disk does exhibit some quirky behavior, but this is not due to errors in the core itself.

CHAPTER 8

DESIGN INTEGRATION

Although much of the design and testing was done on the ML405 evaluation board, the SATA core was developed to be integrated into the design of the DAS board. Data sampled from the two ADCs can be stored on the disk using SATA. Figure X shows how the SATA core was added to the DAS hardware design on the Data FPGA.

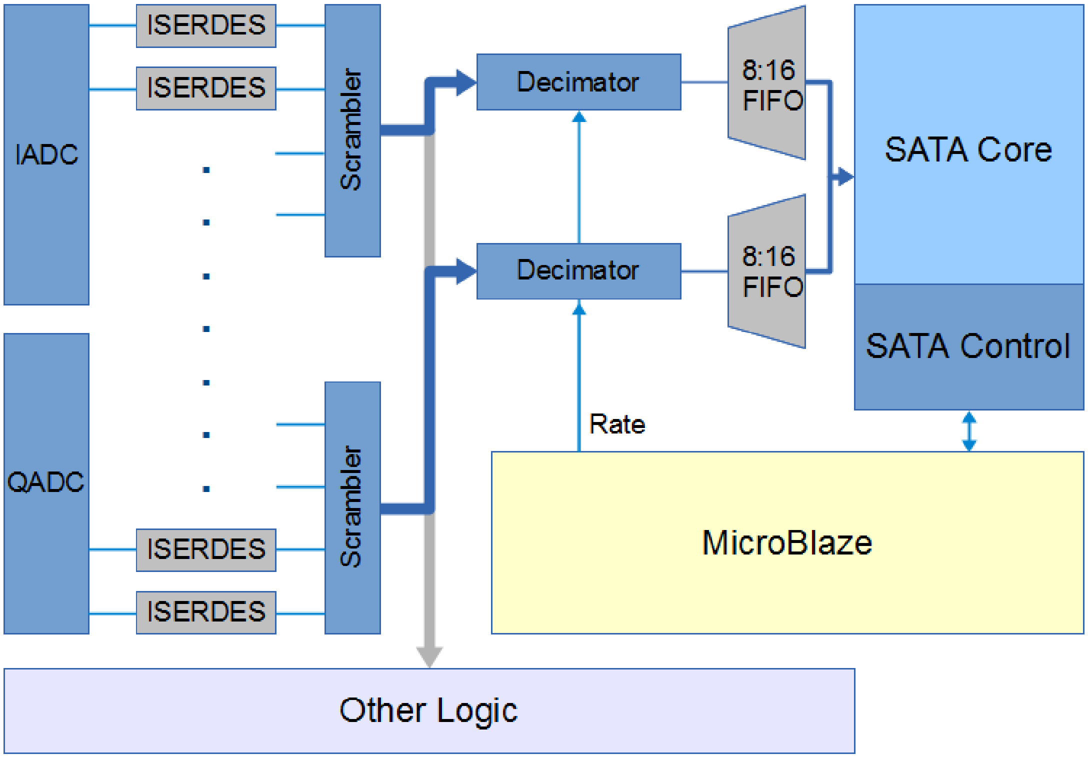


Figure 25: DAS Board Design Integration

Data from each of the ADCs is fed into 1:8 deserializers, which are then reordered so that each byte corresponds to one data sample. 32 samples arrive on each 93.75MHz clock cycle, a rate of 3Gsamp/sec, for each ADC. Unfortunately, this rate is too high to store using SATA, so this data is fed into a decimator. The decimator selects samples at a rate chosen in software, with a minimum rate of 32 and a maximum of 65535. These samples are fed into a clock-domain-crossing FIFO, which connects to the SATA core. A control module handles movement of data between the ADC FIFOs and the core, as well as the triggering of SATA commands. It interfaces with the Microblaze processor via a set of control and status registers. In software, a set of functions are available to access these registers.

8.1 SATA II

The Virtex-4's RocketIO MGTs are capable of supporting a line rate of 3 Gb/s, used by the second generation of SATA. Operation at this rate requires a 300MHz reference clock, which unfortunately the ML405 does have. The DAS board, however, has oscillators that can be easily replaced. Thus, a 300MHz oscillator can be used to drive the core at SATA II speeds.

The results of testing the hard drives at SATA II speeds is shown in Figure 26 below. Interestingly, the performance for the Seagate and WD drives is actually worse. Both drives send many HOLD primitives, reducing performance. The SSD is able to achieve a maximum rate of 182MB/s, greater than the highest rate supported by SATA I. Thus, the use of SATA II is considered a success.

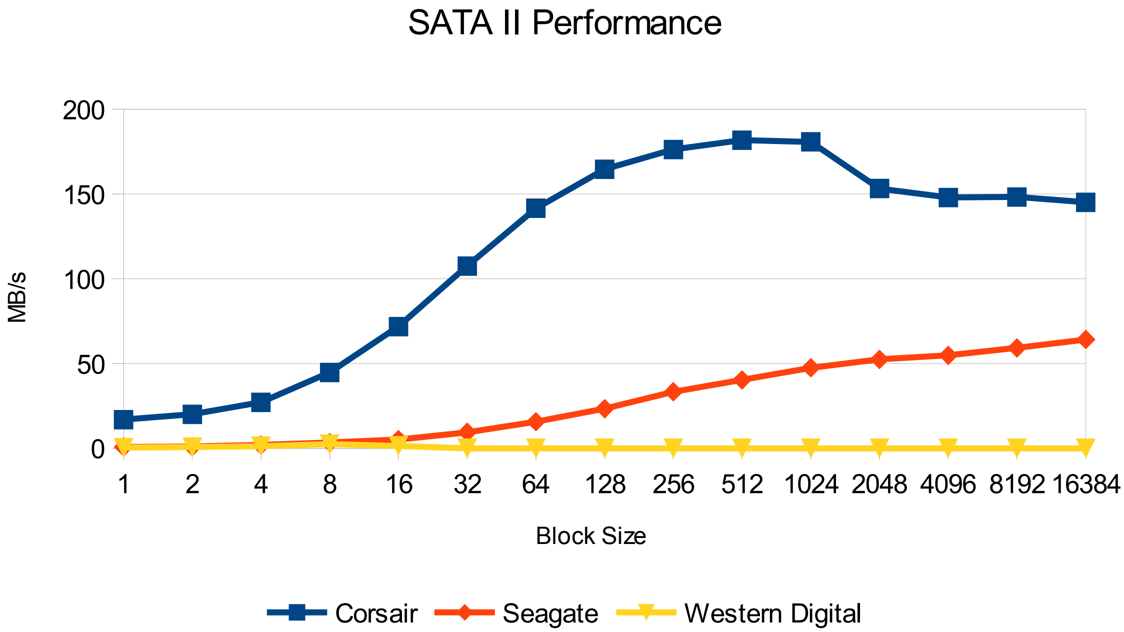


Figure 26: SATA II Speed Test Results

One caveat is that the SATA core does not include speed negotiation logic. Thus, if a SATA I drive needed to be used, the core would have to be re-synthesized in SATA I mode to interface with this disk properly.

8.2 Data Acquisition Testing

The newly integrated SATA core was tested on the DAS board to see if data from the ADCs could be properly stored. Figure 27 shows data from a 62MHz sinusoid signal that was captured and stored on the Corsair Nova.

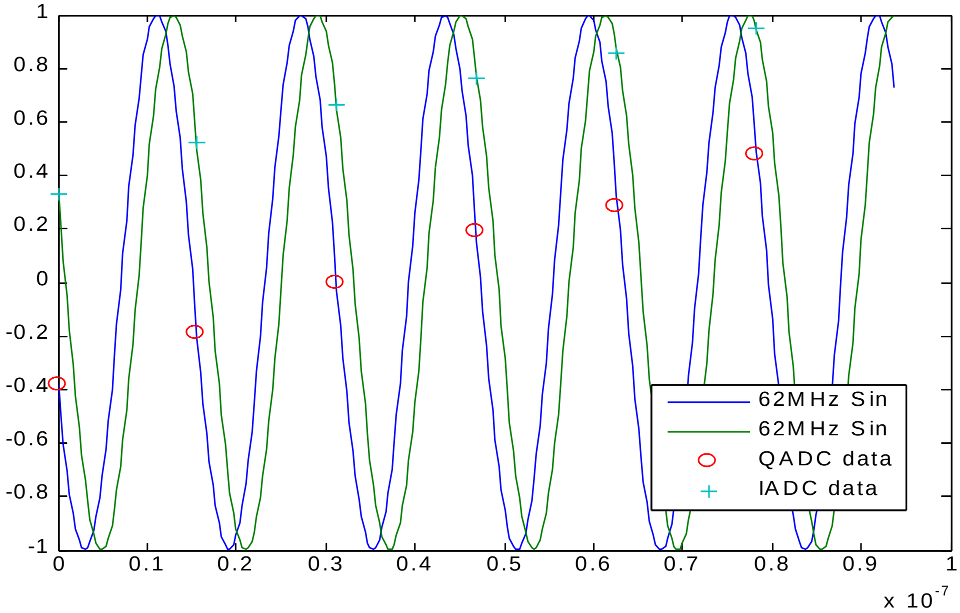


Figure 27: Stored Data

Ideally, a continuous stream of data could be captured and stored to the disk. However, this is not possible at a high transfer rate. Although writes to the disk can average a very high speed, due to caching on the disk these speeds are not constant. The disk controls the rate of any given transfer through the use of HOLDs or by waiting to send a DMA Activate. In testing, we find that the throughput for individual transfers can be as low as 0.0025 MB/s. This was found by looking at the average transfer rate for the operation during which the post-decimator FIFOs overflow. The results of this test are shown in Figure 28.We can assume that this corresponds to the disk emptying its cache, so it must “pause” the transmission until it can ready itself again.

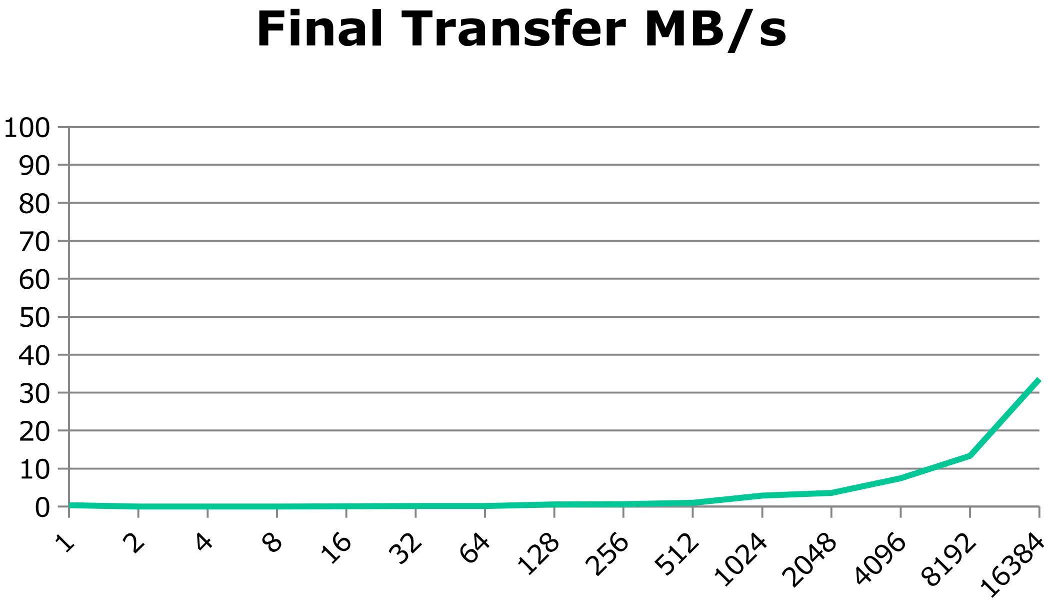


Figure 28: Transfer Rate when FIFOs Overflow

The solution to this problem would be to buffer the data. Unfortunately, there is not enough on-chip memory on the FPGA to do so. The length of the “pause” varies around an average of 127ms, but the longest observed was 528ms. As an example, if we wanted to continuously collect samples at a rate of 6MB/s (a decimation factor of 1000), the FPGA would potentially need to buffer 3MB of data. There is only about 1.2MB of memory on the chip [13], including the memory used by the SATA core and other logic. Thus, off-chip memory would be needed. The following table shows the amount of buffer memory that would be necessary to continuously sample at various rates, assuming a pause time of 528ms. However, even more memory would be recommended, since it is possible that the pause time could exceed this.

Decimation Rate	**Effective Sampling Rate (MB/s)**	**Necessary Buffer Size (MB)**
34	180	95.04
50	120	63.36
100	60	31.68
200	30	15.84
400	15	7.92
600	10	5.28
1200	5	2.64
3000	2	1.056
6000	1	0.528
12000	0.5	0.264

Table 8: Necessary Buffer Sizes

A test was conducted to find the minimum decimation rate that would allow for continuous sampling with the resources available on the DAS board. In this test, data was sampled continuously for a minimum of 1 minute. If successful, the decimation rate would be lowered, and on an overflow, it would be increased. This was done in a “binary search” pattern, with starting bounds of 50 and 20000 for the decimation rate, until the lowest decimation rate that did not overflow was found. This test was then conducted 22 times. The results are shown in Figure 29. The average minimum successful decimation rate is 8401, resulting in an effective sampling rate of 0.714 MB/s.

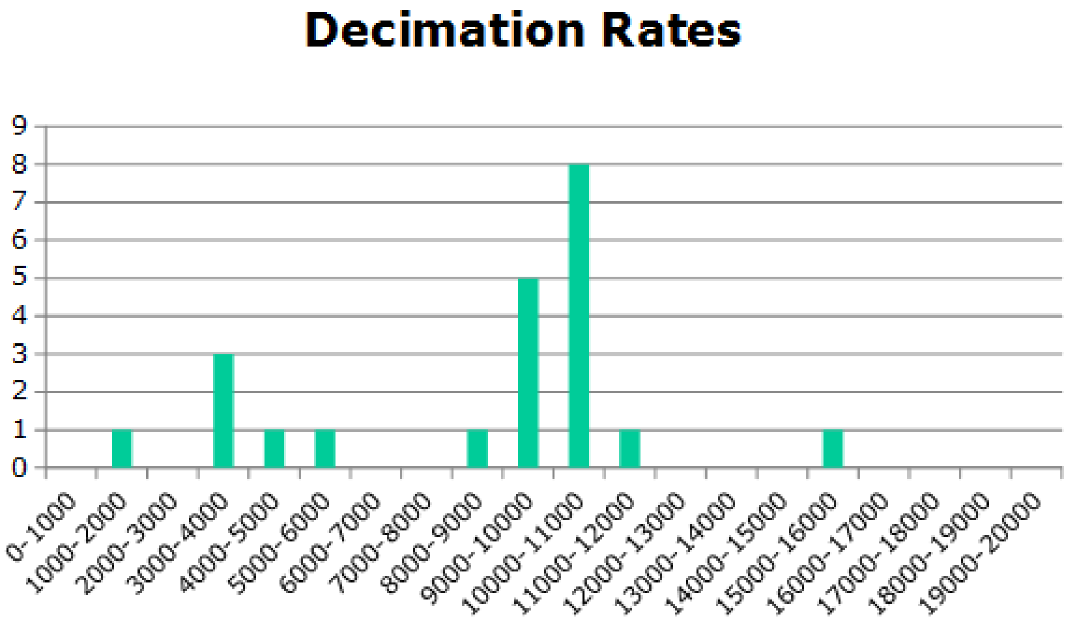


Figure 29: Minimum Decimation Rates

We conducted another test to demonstrate that a large amount of data could be written to the SATA disk correctly. In other words, we tested to see if any data samples were dropped or missing from the data written to the disk.

For this experiment, a 2.3MHz sine wave was created on a signal generator and fed into the ADCs. The data was then collected with a decimation rate of 400 until an overflow occurred (a total of 84MB of data was collected). This data was then read in MATLAB and subtracted from a MATLAB-generated sine wave. If any samples are dropped, we would expect these residuals to have a large “jump”, because the captured signal would suddenly be out of phase with the generated wave. The results of this experiment are shown in Figure 30.

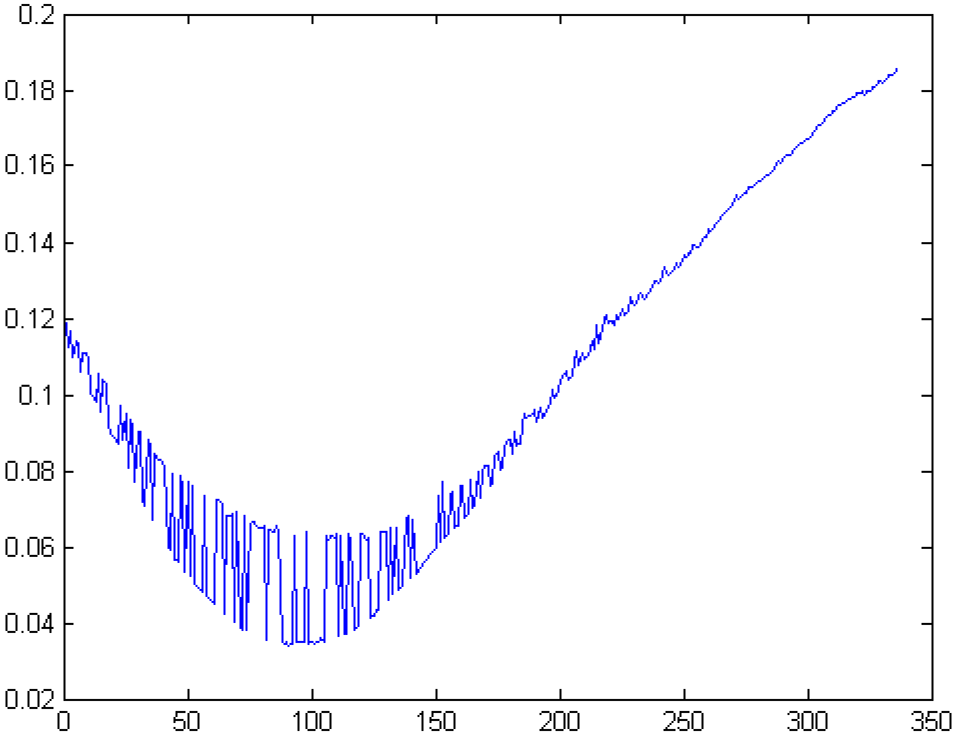


Figure 30: Stored Data Residuals

The sine wave frequency used to create these residuals was 2.30000315MHz. The residuals vary due to a slight frequency mismatch between the generated sine wave and the data. However, there is no “jump” in these residuals, so we can conclude that no samples are being dropped. Thus, the SATA core is successfully able to store all received data even in large amounts.

CHAPTER 9

FUTURE WORK

Although the SATA core is currently very functional, there are still a number of features that could be added.

One feature currently not supported is power management. The SATA protocol includes the ability to switch between power states, but the core currently does not support it. This feature would conserve energy for applications where that is important, such as on a satellite.

Another feature would be speed negotiation support, allowing the core to be fully compatible with both SATA II and SATA I disks. This would require some extra steps during the OOB initialization sequence.

Command queuing could also be implemented. This is a feature wherein multiple pending operations can be issued, and the disk can respond to them in any order. This feature can lead to significant performance improvements in spinning disks by reducing the seek time [10]. This feature would be fairly complicated to implement, as the core would need to keep track of outstanding operations and transmit or receive data out of order.

If the core was to be integrated into an embedded Linux system, it could benefit from an ATA interface. As mentioned in section 2.2, the Command Layer was designed to emulate the parallel ATA interface. By implementing the set of shadow registers and adding support for other ATA command types, the core could be easily utilized by a Linux system, since ATA drivers are readily available. A complete file system could be created on the disk.

Recall that the ESS design included a DSOTDM module to find the correct OOB threshold. The current design does not have such a module, and thus the correct threshold needs to be found manually. A similar module could be incorporated into the new design to make it easier to use on other boards.

Finally, the SATA core could be made more robust by testing on other drives. Although the current error rate is very low, there are certainly hidden cases where an error could appear. Also, the incoming data could be verified. As is, the core does not check the CRC of received FISes. Doing so would ensure correctness of receive operations.

CHAPTER 10

CONCLUSION

A SATA core has been created that is compatible with Virtex-4 FPGAs. This core utilizes the RocketIO high-speed transceivers to achieve line rates of 1.5Gb/s or 3.0Gb/s. The core implements all layers of the SATA protocol so that it can communicate with any SATA-compatible hard disk. It presents a simple Read/Write interface so that it can be easily integrated with other modules. This SATA core will be released as open-source, and will be the first freely-available SATA core for the Virtex-4. The core has been shown to work on the ML405 board and a custom board developed for NASA, where it can be used to store a very large number of samples. It has been tested against three different hard drives from three different manufacturers to ensure that the SATA protocol has been implemented correctly.

BIBLIOGRAPHY

[1] "Serial ATA: Meeting Storage Needs Today and Tomorrow". SATA-IO. http://www.serialata.org/documents/SATA-Rev-30-Presentation.pdf, Jun 2009.

[2] K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications*. Hillsboro, OR: Intel Press, 2003.

[3] Justin Lu; P. Siqueira; V. Vijayendra; H. Chandrikakutty; R. Tessier, "Real-Time Differential Signal Phase Estimation for Space-Based Systems using FPGAs," *Aerospace and Electronic Systems, IEEE Transactions on*, vol.49, no.2, pp.1192,1209, Apr 2013. doi: 10.1109/TAES.2013.6494407

[4] V. Vijayendra, "Design and Testing of a Prototype High Speed Data Acquisition System for NASA," M.S. Thesis, ECE, University of Massachusetts Amherst, 2010.

[5] "ML405 Evaluation Platform: User Guide." Xilinx, UG210, Mar 2008.

[6] D. Anderson, *SATA Storage Technology*. Colorado Springs, CO: Mindshare Press, 2007.

[7] S. Tam and L. Jones, "Embedded Serial ATA Storage System". Xilinx Application Note 716. http://www.xilinx.com/txpatches/pub/applications/xapp/xapp716_release.zip, Oct 2006.

[8] "Serial ATA I/II Host Controller (SATA_HI)." ASICS World Services. http://www.xilinx.com/publications/3rd_party/products/ASICSWS_SATA_H1.pdf, May 2008.

[9] "LogiCORE IP Virtex-4 FX FPGA RocketIO Multi-Gigabit Transceiver Wizard v1.7: Getting Started Guide." Xilinx, UG246, Apr 2010.

[10] A. A. Mendon; B. Huang; R. Sass, "A High Performance, Open Source SATA2 Core," *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, vol., no., pp.421,428, 29-31 Aug. 2012. doi: 10.1109/FPL.2012.6339139

[11] "Virtex-6 FPGA GTX Transceivers: User Guide." Xilinx, UG366; Jul 2011.

[12] "Virtex-4 RocketIO Multi-Gigabit Transceiver: User Guide." Xilinx, UG076; Nov 2008.

[13] "Virtex-4 Family Overview." Xilinx, DS112; Aug 2010.